ビデオゲーム記述をドメインとする プログラミング言語

筑波大学審査学位論文(博士)

２０１１

西森 丈俊

筑波大学大学院ビジネス科学研究科企業科学専攻

目 次

第1章　研究の背景と目的

1.1　はじめに

ビデオゲーム産業は，情報技術の進歩を背景に短い間に急速に成長し，現在も成長を続けているコンテンツ産業の１つである．特に「ファミリーコンピュータ」にはじまる家庭用ビデオゲーム機は，その手軽さ等の理由から，現在では，一般的な娯楽として定着するほど普及してきた．

ビデオゲームは，ハードウェアと，そのハードウェア上で動作するソフトウェアの２つから成る．ソフトウェアがハードウェアと分離していることにより，ユーザーは，ハードウェアを買い換えなくとも，ソフトウェアを購入することで新しいゲームを遊ぶことができる．そのソフトウェアの充実のため，ハードウェアメーカーは積極的にソフトウェアメーカーの参入を促進し，多種多様のゲームソフトウェア製品が開発販売された．このことと，ビデオゲームが手軽な娯楽製品であるという理由から，ビデオゲーム産業は大きく成長を続けてきた [新宅 03] [1]．

ビデオゲームソフトウェアは面白さを提供することを目的とするが，面白さの定義は難しく，実装前に十分な設計をすることが難しい．そのため，ビデオゲームソフトウェアは，頻繁なトライアンドエラーを繰り返しながら開発される．しかし，産業の成長に伴いビデオゲーム開発プロジェクトの規模も大きくなってきたことから，開発スタッフ間でやり取りしながらトライアンドエラーを進めることは難しくなっている．このことは，面白さを十分に達成できない要因となる．

ゲーム開発では，ゲームの企画立案や設計はゲームデザイナーにより行われる．ゲームデザイナーが直接トライアンドエラーを行うことができれば，開発スタッフ間のやり取りは軽減されるので，開発はより円滑に進められ，目的とする面白さを達成することも容易となる．そのため，近年では，トライアンドエラーを容易にするためのツールとして，ゲームシステム記述ツールが広く利用されるようになっており，ゲームデザイナーが直接トライアンドエラーを実施することも可能となってきている．

しかし，現在のツールは十分な機能を提供していないので，ゲームデザイナーだけで実施できるトライアンドエラーは限定され，目的とする面白さを達成するために必要なトライアンドエラーを十分に行うことが難しい．

---

[1] 日本コンピュータエンターテイメント協会 (CESA) ゲーム白書によると，2003 年のビデオゲームソフトウェア出荷額は約 4747 億円であったのに対し [ces03]，2009 年のビデオゲーム出荷額は約 1 兆 244 億円と報告されている [ces09].

そこで，本研究は，ゲームデザイナーが直接トライアンドエラーを行うためのツールとして，トライアンドエラーを必要とするゲームの要素を記述するのに適した，プログラミング言語の提案及び設計と実装を行い，ゲーム開発に適用してその評価を行うことを目的とする．

次節からは，ビデオゲーム開発の特徴や開発方法について議論し，本研究が目的とするプログラミング言語の必要性や，プログラミング言語に求められる機能要件について検討する．

1.2　ビデオゲームソフトウェア開発の特徴

ビデオゲームソフトウェアには次の 3 つの特徴がある．

- 常に「面白さ」を要求される．
- 多様性と変化の激しさに対応しなければならない．
- 製品や開発が急速に大規模化してきている．

娯楽商品であることから，ビデオゲームには,「面白さ」が要求されるが,この「面白さ」を定義することや合理的に評価することは難しい [SZ02]．そのため，他のソフトウェア産業とは違い，制作前に明確な仕様を作ることが難しい．

ビデオゲームが娯楽商品であることから，販売される製品は多様性があり変化が激しい [新宅 03]．一度成功すれば長期にわたって収益を上げることができるビジネスソフトと違い，ビデオゲームは，新規性のある面白さを求める傾向から古い商品は評価されず，商品が 1 回限りで寿命が短い．このため，ビデオゲーム産業は常に新しい製品を提供し続けなければならず，面白さで他の商品と差別化するために多様化し，変化も激しくなる

ビデオゲームソフトウェア開発は近年大規模化している [新宅 03]．情報技術の急速な進歩により，ビデオゲームソフトウェアには，3D グラフィクスや DVD 等の新しい技術的要素が取り入れられ，より高水準で大量のコンテンツを実現できるようになった．そのため，制作にはより多くの開発人員や時間を必要としてきている．

1.3　探索的なビデオゲームソフトウェア開発

ビデオゲームソフトウェアは娯楽製品であることから，常に新しい面白さを要求される．目的とする面白さを達成するために，トライアンドエラーを繰り返したり複数のバージョンを作り比較する等の，探索的な手法で開発が進められるが [新清 02, 新宅 03]，これは次の理由による．

- 制作前に面白さを検証することが難しい．

- 開発プロジェクト内でのイメージの共有が必要.
- 実現するパフォーマンスの予測が難しい.

制作に入る前に製品の「面白さ」を検証することは難しい．ビデオゲームで最も重要な要素である「面白さ」は，定義が難しく曖昧なものであるため，実際に遊ばずに面白いかどうかを評価することできず，開発は面白さを確認しながら進める必要がある [SZ02, Cra82]．このためビデオゲーム開発では，一般的なソフトウェア開発で行われる設計，実装，テストを分離した方法ではなく，頻繁なトライアンドエラーを繰り返す開発手法が用いられている [新清 02, 新宅 03].

また，開発前にイメージした面白さが，製品に反映されるためには，製品イメージを開発プロジェクト内で共有し，開発スタッフが目的とする製品の面白さに向けて開発を進めていくことが必要となる．しかし，面白さの定義が難しいことから，制作がある程度進まなければ製品イメージを共有することも難しい．このことも，制作と検証を何度も繰り返す要因となっている．

パフォーマンス予測の難しさは，ハードウェア上の制約が厳しいことや，グラフィクス等で高度な記述や新規の機能を要求されることが要因となっている．ハードウェア上の制約や要求の高度さのために意図したパフォーマンスが得られるかどうかの予測がしづらく，設計とコーディングを別々のステージとすることができない．このため，設計とコーディングが同時的に行われることが多い [新宅 03].

このように，ビデオゲーム開発では，トライアンドエラーを繰り返す探索的な開発プロセスが不可避となっている.

1.4 ビデオゲーム開発におけるゲームデザイナーとその役割

ビデオゲーム開発は，企画の立案やゲームのおおまかな設計を 1 人から数人で始める．ゲームの概要が固まってから，小規模なプログラムやグラフィクス等の素材の制作を開始するためにスタッフが増員される．ゲームの基本的な部分でのトライアンドエラーを経て開発が進み，開発の方法や流れが決定すると，完成した製品とするのに必要となる多くのコンテンツを制作するために，さらに多くのスタッフが増員される．多くのコンテンツが追加され，ゲーム全体の面白さの調整のためにトライアンドエラーも頻繁に繰り返される [新宅 03, 松原 03].

ビデオゲーム開発プロジェクトは主に次のスタッフから構成される.

- ゲームデザイナー
- プログラマー
- グラフィックデザイナー

- サウンドクリエーターやデバッガー等，その他のスタッフ

企画立案やゲームの設計，開発ディレクションを担当するスタッフのことをゲームデザイナーと言う[2]．多くのゲーム開発において，ゲームデザイナーは開発プロジェクトの最初から最後まで参加し，ゲーム内容について直接の担当者であることから，他の開発スタッフとのコミュニケーションや連携作業が多い [新宅 03]．自らプログラミング言語やその他のツールを用いて試験実装を手掛ける場合もある．

ビデオゲームのプログラム制作を担当するスタッフをプログラマーと言う．プログラムは，ゲームアプリケーションの動作を実装する基礎的な部分であることから，ゲームデザイナーとのやりとりも多く，トライアンドエラーも，ゲームデザイナーと連携して頻繁に行う．ゲーム開発におけるプログラマーは，3D グラフィクス等の新しい技術を用いた実装や，ハードウェアの制約内でも十分なパフォーマンスを得るために，高度なプログラミング技術を有することが求められる．

グラフィックデザイナーは，ゲームに登場するキャラクタの 3D 素材や，そのアニメーションを制作するスタッフのことを指す[3]．より大量のコンテンツをゲームアプリケーションに搭載できるようになってきたことから，グラフィックデザイナーの作業量も増大している．グラフィクスがゲームの世界を適切に表現していることや，コンテンツを結合したときの統一性の確認のために，ゲームデザイナーと連携して修正を繰り返す作業も多い．

ゲーム開発プロジェクトには，上記の他に，音楽や効果音を担当するサウンドクリエーターや，テストを専門に行うデバッガー[4] 等がいる．

ゲームデザイナーはゲーム内容についての直接の担当者であることから，開発の中心的な役割を果たす．面白さの評価や実装の適切さの検証はゲームデザイナーが担当するので，ビデオゲーム開発は，ゲームデザイナーが主体となって，他の開発スタッフと連携しながらトライアンドエラーを行う形で進められる (**図 1.1**)．

ゲームデザイナーには，Action Script[Ado] 等のプログラミング言語を用いた小規模なプログラムの制作や，グラフィクスツールを用いた簡単な作画や動画制作を行うことができる者が多い．これは，制作前には遊んで試してみる事のできない面白さを開発スタッフに伝えるためには，表現しようとする面白さを，コンテや動画，簡単なプログラム等を使って具体的に示すことが有効であることと，ゲーム設計には，ゲームアプリケーションを構成するプログラムやグラフィクス等について幅広い知識を必要とすることが理由である．また，プログラミングの経験が無くとも，ゲームを分析して要素を抽出し，それをプログラムとして動作するよう，矛盾なく設計しなければなら

[2]日本国内では「企画」と呼ぶことも多い．

[3]グラフィックデザイナーは，単に「デザイナー」と称される場合もあるが，本論文ではゲームデザイナーと区別するために「グラフィックデザイナー」を使用する．従事する制作内容により「アニメーター」や「モデラー」といった異なる呼び方をする場合もある．

[4]一般的には，デバッガーとはプログラムのバグを見つけるためのツールのことであるが，ゲーム業界では，アプリケーションのテストを行い不具合や感想等の報告をする担当者を指すことが多い．

ないことから，ゲームデザイナーには，プログラムの動作や仕組みについて詳しい知見を持つことが求められる．

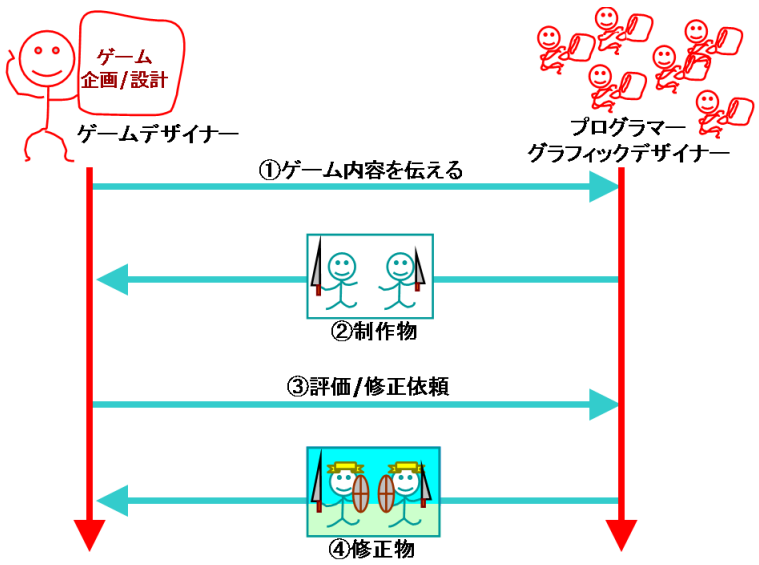

図 1.1: ビデオゲーム開発の流れ

1.5 トライアンドエラーを必要とするゲーム要素

娯楽製品であるビデオゲームは，定義や制作前の評価が難しい「面白さ」を提供しなければならない．このため，その開発は，ゲームデザイナーが中心となり，他の開発スタッフと連携してトライアンドエラーを何度も行いながら進められる．

トライアンドエラーを最も必要とする，ビデオゲームの面白さの要素としては，次のものが挙げられる [SZ02]．

- インタラクション性
- プレーヤーやゲームキャラクタ同士の競合性
- 不確実性
- ゲームルール

インタラクション性とは，ユーザーの操作がゲームに反映されることを言う．例えば，ユーザーがパッドやスティックなどの入力デバイスを利用してゲーム中のキャラクタを操作する場合，ゲーム中のキャラクタはユーザーの操作を反映した動作する．ユーザーの操作が，直接または間接的にゲームに

反映されることで，ユーザーはゲームを進めている実感を得ることができる．インタラクション性はビデオゲームの面白さの要素の１つである．

競合性とは，目的達成のためにゲーム中に発生する競り合いや奪い合いのことである．例えば，ゲーム中のキャラクタ同士の倒し合いは，ゲームの目的として設定されることが多い．また，ゲームの目的がゴールまでたどり着くことである場合，それを邪魔する敵キャラクタは，ユーザーの操作するプレーヤーキャラクタが前に進むのを阻止することを目的とし，プレーヤーキャラクタと敵キャラクタの間に目的の違いによる競合が発生する．これらの競合性の要素も，ゲームの面白さにとって重要である．

不確実性とは，ユーザーが同一の操作をしても異なる結果がでることをいう．不確実性は，ユーザーに新しい展開や予測できない結果を期待させることを可能にさせ，ビデオゲームの面白さの１つとなる．

ゲームルールとは，前述の３つの要素にも関係するが，ゲームキャラクタの挙動の仕組みや制限，キャラクタ間の関係や，獲得できるポイント，相手キャラクタに与えられるダメージ量等を指す．ユーザーはこのゲームルールに従ってゲームを進行させる．ゲームルールが十分に練られたものであれば，ユーザーはゲームの進行に納得し，繰り返し遊ぶ動機にもなるが，不合理な点や曖昧な部分があると不正や不公平を感じさせ，ゲームへの興味を削ぐ原因となる．

上に列挙したインタラクション性，競合性，不確実性，ゲームルールのことを，グラフィクスデザインやサウンドなどの要素と区別して「ゲームシステム」と呼ぶ．ゲームシステムは，面白さの中核的な要素であり，定義が難しいため制作に先立って十分な設計ができず，実際に遊んでみなければ評価をすることも難しい．このためゲームシステムは，開発時に最もトライアンドエラーを必要とする部分である．

ゲームシステムはゲームデザイナーが設計するが，実装にはプログラマーによるプログラミング作業を必要とする．ゲームシステムを面白いものとするためには，十分なトライアンドエラーが必要であるため，ゲームデザイナーとプログラマーの間では，ゲームシステムについて理解を深めるための議論や，テストや評価を反映するためのやりとりが何度も繰り返されるが，このやり取りは手間や時間を必要とする．

1.6　ゲームシステム記述ツールを用いた開発

ゲームシステムは面白さの重要な部分であることから，特に多くのテストや修正を必要とする．しかし，次の理由から，大規模化していく製品の開発プロジェクトで，トライアンドエラーを何度も繰り返すことは難しい．

- 高度で複雑なプログラムは，トライアンドエラーを難しくする．
- ゲームシステム担当者であるゲームデザイナーが直接実装やテストを行うことができない．

ビデオゲーム開発では，高度なグラフィクスアニメーションや大量のコンテンツ処理のために，プログラム実行時の高速性や効率性が求められる．そのため，開発にはプログラミング言語として，C/C++言語 [KR89, Str98] やアセンブリ言語を用いることが多い．これらのプログラミング言語は，汎用的で実行速度や効率を達成するのに適している反面，ゲームシステムの記述に適した機能が無く，ゲームプログラミングについての十分な知識や経験が要求され，プログラムも高度で複雑なものになる．そのようなプログラムは修正することも容易ではない．このことが，トライアンドエラーを繰り返すことを難しくする．

さらに，大規模化する開発においては，アニメーションやコンテンツの処理を実装するプログラマーの制作作業量も増大するので，より細かく作業の分担をしなければならない．しかし，前述の C/C++言語やアセンブリ言語でゲーム開発を行う限り，ゲームシステムの実装にはこれらの言語によるプログラム記述が必要となるので，プログラミングに精通していないゲームデザイナーは直接実装や修正を行うことが難しい．このため，ゲームシステムの実装や修正はゲームデザイナーとプログラマーの間での共同作業となり，ゲームシステムについて理解を深めるための議論や，テストや評価を反映するためのやりとりが何度も繰り返される．このことは，細かなテストや修正を何度も繰り返す上での障害となる．

そこで，現在のビデオゲーム開発では，ゲームシステムの実装に C 言語やアセンブリ言語を用いず，よりゲームシステム記述に適したプログラミング言語やツールを用い，ゲームシステム部分のテストや修正を容易にする開発手法が用いられる [Daw02a, igd09, VR02, Gro10]．これらのプログラミング言語やツールのことを，本論文では「ゲームシステム記述ツール」と呼ぶ．ゲームシステム記述ツールを導入することにより，次の効果が得られる．

- トライアンドエラーを繰り返すことが容易になる．
- ゲームデザイナーが直接実装やテストを行うことができる．
- 製品イメージの共有を促進させる．

ゲームシステム記述ツールは，C/C++やアセンブリ言語と比較して，よりゲームシステム記述に適した機能を持つことから，ゲームシステム部分のテストや修正がしやすくなり，トライアンドエラーを繰り返すことが容易になる．

また，ゲームシステムの記述に適した記述ツールを用いることで，汎用的なプログラミング言語と比較して高度なプログラミング技術無しに，ゲームシステムの開発が可能になるので，小規模なプログラム開発経験やプログラムの動作についての一般的な知見のみを持つゲームデザイナーが直接ゲームシステムの実装やテストを行うことができるようになる．ゲームシステム記述ツールを導入しない場合，**図 1.2** の上部分のように，アプリケーション全ての実装はプログラマーが行うが，導入する場合，ゲームデザイナーがゲー

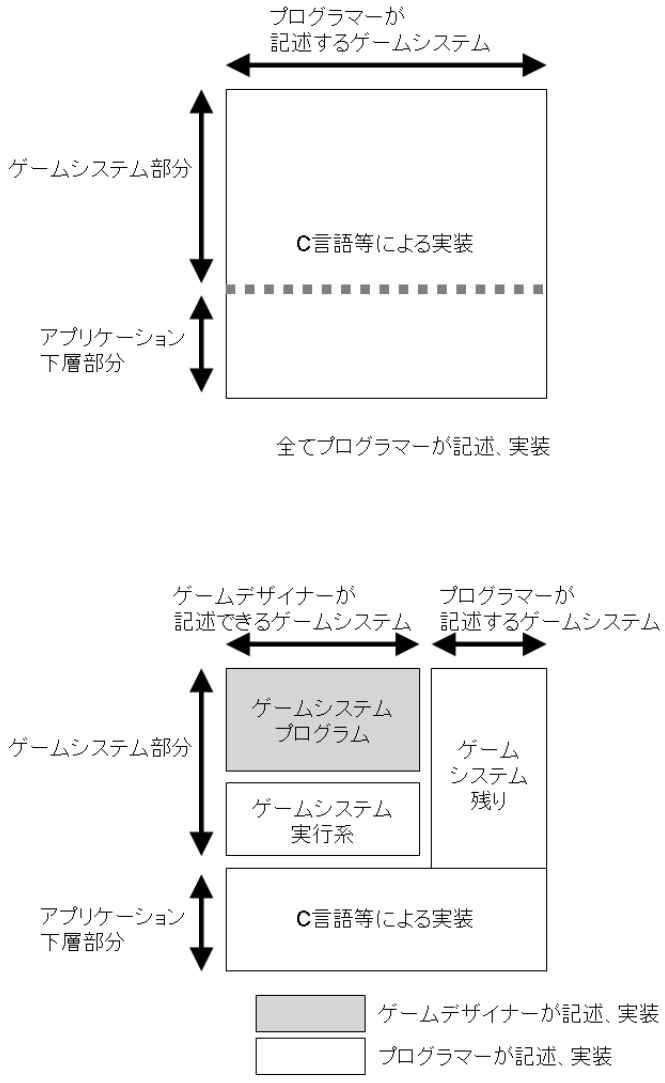

図 1.2: 記述ツールを使用しない場合 (上) と使用した場合 (下) のソフトウェア構造と作業分担図

ムシステム部分の実装を行うことが可能になる．**図 1.2** の下部分はその分担を表したものである．ゲームシステムをゲームデザイナーが実装することにより，プログラマーはゲームシステムについての作業が減り，ゲームデザイナーはプログラマーとの連携が無くともゲームシステムについての開発を進めることができる．この結果，開発プロジェクトはゲームシステムについて，トライアンドエラーを繰り返したり複数のバージョンを作り比較するような探索的なプロセスで開発を進めやすくなる．

また，開発プロジェクト内で，製品イメージの共有を図るための手段として，ゲームシステム記述ツールを利用することもできる．これは，ゲームシステム記述ツールを使用することで，ゲームデザイナーが自分の持つゲームイメージを部分的にでも直接表現でき，他の開発スタッフが製品イメージを認識しやすくなるからである．開発スタッフ間で製品イメージの共有が進むと，個々の開発者がより製品に適したコンテンツの制作をしやすくなり，修正や作業のやり直しが減る [竹田 00]．これにより，開発時間や作業人数等の開発コスト要因を抑えることができる．

ビデオゲームソフトウェア開発の大規模化は，開発コストの増大につながるので，製品はコストに見合ったより大きな売上を出すことが求められる．売上のためには，多くのユーザーに受け入れられる面白さを持つ製品にしなければならないので，十分なトライアンドエラーが必要である．しかし，開発スタッフが多い大規模な開発では作業は細分化され，テストや修正を何度も行うことが難しくなり，製品イメージを共有するためのコミュニケーションにもより多くの時間を必要とする．そのため，トライアンドエラーを容易にし，ゲームデザイナーによる直接的な実装も可能にし，イメージの共有にも貢献するゲームシステム記述ツールは，近年ビデオゲーム開発において広く利用されるようになっている．

1.7 現在利用されているゲームシステム記述ツール

現在多くのビデオゲーム開発において，記述ツールが積極的に導入され，ゲームデザイナーが記述ツールを使ってゲームシステムの制作を行なっている．

ゲームの面白さの中核となるゲームシステムを容易に記述可能にするゲームシステム記述ツールは，高度なプログラミング技術無しに，ゲームシステムの制作を可能にする．これらのツールは，ある程度のプログラミング技術や，プログラムの構造及び動作の仕組みについての知識を持つゲームデザイナーにも利用可能であり，プログラマーの協力無しにゲームシステムの制作を進めることができる．ゲームデザイナーが直接制作に携わることができることから，プログラマーとのやり取りも減り，トライアンドエラーが容易になり，開発スタッフ間での製品のイメージ共有も容易になる．

2 章でくわしく述べるが，現在，ビデオゲームや類似するアプリケーションを記述するツールとしては，様々なものが提供，利用されている．これらのツールを大きく次の 4 つに分類する．

- ビデオゲーム開発を目的としたプログラミング言語
- 統合的なビデオゲーム開発環境
- 個々のゲームの開発専用に作られた記述ツール
- 類似するその他のツール

ビデオゲーム開発を目的としたプログラミング言語とは，2 章で述べる Lua[IdFF96] のように，ゲームアプリケーションに組み込まれて利用されることを目的としたプログラミング言語や処理系である．製品やプラットフォームを選ばず，ゲームアプリケーションへの組み込みが容易で，処理系の実装も軽量という特徴がある．言語を利用した記述方法や組み込み方法などの文書や情報が予め用意されていることから，全く何もないところから記述ツールを開発する必要がない．また，他の汎用的なプログラミング言語にはない，ゲームシステムの記述を容易にするための機能を提供するものもある．

ビデオゲーム開発を目的としたプログラミング言語は，多くのビデオゲーム開発で利用されている．これは，新規にプログラミング言語の設計や実装をすることが簡単ではなく時間がかかり，既に実装として存在するものを利用する方が導入が容易であることと，プログラミング言語の学習のために必要とするドキュメントや情報が利用可能であることが理由である．開発の早い段階から導入でき，学習のための情報が用意されていれば，ゲームデザイナーは早期からゲームシステムの実験やテストをしやすくなる．

統合的なビデオゲーム開発環境とは，ビジュアルインターフェースを用意し，グラフィカルに開発を進められ，グラフィクスやサウンド等のデータもすぐに利用可能なツールのことを指す．これらのツールは操作が容易な上，ゲームに使用するデータの使用や管理も容易に行える．また，ゲームアプリケーションプログラムを出力するので，プログラミング言語を用いた記述無しにある程度のゲームアプリケーション開発が可能である．しかし，グラフィカルインターフェースだけでは詳細なキャラクタの動作や複雑なゲームルールの実装が難しい．そこで，これらのツールにはプログラミング言語を用いた記述機能も用意されている．

統合的なビデオゲーム開発環境では，高度なプログラミング技術を必要とするグラフィクスやサウンド，衝突等の処理をするプログラムがあらかじめ用意されているので，プログラミングを作業を減らすことができ，開発時間の短縮につながる．また，ビジュアルインターフェースを用いて，容易にゲーム場面のキャラクタの構成や設定をできることと，ある程度のプログラミング技術があれば，細かな実装も行えることから，ゲームデザイナーがゲームシステムの実装に直接関わることも可能になる．これらの理由から，統合的なビデオゲーム開発環境は，近年広く利用されるようになってきている．

ビデオゲーム開発では，個々のゲーム専用に新しい記述ツールを制作することも少なくない [西森 03]．これらの，個々のゲーム専用の記述ツールでは，ゲーム内容を記述しやすくしトライアンドエラーを容易にするために，他の

ツールには無い，目的とするゲーム独自の記述方法が用いられている．また，実装も目的とするゲームプログラムに適したものとなっていることから，実行速度や記憶領域の効率性等においても他の記述ツールより有利なことがある．これらの記述ツールも，詳細な記述を可能にするために，独自のプログラミング言語やそれに近い記述方法を提供する．

個々のゲーム専用に記述ツールを設計実装することで，他の記述ツールにはない独自の設計や機能を取り入れることができる．あるゲームの開発で，頻繁なトライアンドエラーが予想される部分については，その記述が容易な設計を記述ツールに導入することで，開発を円滑に進められるようになることが期待できる．また，他の開発への再利用を考慮する必要がないことから，開発プロジェクト内のゲームデザイナーに合わせた設計にすることもできる．これらの理由から，個々のゲーム開発で独自の記述ツールを設計開発することは少なくない．

前述の 3 つ以外にもビデオゲームに類似したアプリケーションの制作が可能なものが存在する．プログラミング教育用に開発されたツールは，興味を持たせるために簡単なアニメーションやゲームの制作が可能である．また，エージェントシミュレーションやアニメーションオーサリングツールは，ビデオゲームに類似したアプリケーションの制作も可能である．これらのツールは，グラフィカルなインターフェースを用意しているものが多いが，より詳細な記述を可能にするためにプログラミング言語による記述機能を提供するものもある．

1.8 ゲームシステム記述ツールに求められる機能

ビデオゲーム開発では，ゲームシステム記述ツールは，ゲームデザイナーとプログラマーの共同作業を軽減し，トライアンドエラーを容易にするために導入されるが，記述ツールがゲームシステム記述に適した機能を十分に提供できなければ，トライアンドエラーを十分に行うことができず，開発を円滑に進めることも難しくなる．

そこで本節では，トライアンドエラーを容易に進めるために必要な，ゲームシステム記述ツールが提供するべき機能について議論する．

記述ツールに求められる機能としては，次の二つを挙げることができる．

- 記述ツールはゲームシステムの記述に適したプログラミング言語を持つこと．
- 単一のビデオゲームだけでなく，複数のゲーム開発にも利用できること．

ゲームシステム記述ツールは，ゲームシステムの記述に適し，トライアンドエラーが容易なものでなければならない．さらに 1.5 節で述べたゲームシステムの要素には， 数値の列挙やグラフィカルなユーザーインターフェースだけでは記述の難しい要素が多いことから，高い記述性を持つプログラミン

グ言語による記述機能を持つことが望まれる．例えば，アクションゲームでは，ゲーム中のキャラクタの攻撃や防御によるダメージやリアクション等の処理に，複雑で且つ多数の条件や計算式を必要とする．また，多くの敵キャラクタがプレーヤーキャラクタに対して組織立って攻撃するような処理を記述するには，敵キャラクタ同士の位置や体力等の情報の収集と，その情報に応じた挙動の変化が必要である．高度なインタラクション性や競合性を達成するために,「攻撃されてダウン中．移動不可だがボタンによる特殊攻撃が可能」「連続ダメージ回避のために暫く攻撃無効状態だが，攻撃以外の挙動は通常状態と同じ」等の複雑な条件判定や処理が必要となることもある．

これらのゲーム要素の記述は，グラフィカルなインターフェースを用いて図や数値で表現することが難しく，計算式や制御構造等を利用可能なプログラミング言語を用いるのが適切である．したがって，プログラミング言語は，ゲームシステムのトライアンドエラーを容易にするために，ゲームシステムの記述に適した機能を持つことが求められる．

前節で説明した既存の記述ツールの多くが，プログラミング言語を使った記述を機能として提供しているのは，より高い記述性を備えることで，ゲームシステムの実装を可能にするためである．

また，前節で述べた個々のゲーム開発専用の記述ツールのように，特定のゲームアプリケーションに特化した設計とすると，異なるゲーム開発に適用できず，新しい開発では再度記述ツールの制作からはじめなければならないため，開発コストに影響がある．さらに，あるゲームの実装には有用な記述機能が必ずしも他のゲームシステムでも有効とは限らない．このため，記述ツールとして使用するプログラミング言語は，特定のゲームに特化せず, 汎用的な形で，1.5 節で述べた要素の記述に便利な機能を持つことが望まれる．

1.9　ゲームシステム記述に適したプログラミング言語の要件

前節では，ビデオゲーム記述ツールは，ゲームシステムの記述に適し，特定のゲームに特化しない，プログラミング言語による記述機能を持つ必要があることを述べた．本節では，そのプログラミング言語に求められる要件について議論する．

筆者の開発経験では，ゲームシステムを制作する上で，自律したゲームキャラクタの挙動と複数のキャラクタ間に適用されるゲームルール実装は，開発が進むに従って記述が大きくなりやすく，テストや修正も多く必要とする．しかし，記述が大きくなるとプログラム構造も複雑になり，可読性も悪くなるので，追加や修正は容易ではなくなる．従って，ゲームキャラクタの挙動とキャラクタの間に適用されるゲームルールの記述性に優れることが必要であると考える．

以下で詳しく説明するが，ゲームキャラクタの挙動は，状態遷移処理や時

間に沿った処理として自然に表現できる．また，自律した処理は，他の処理とは独立した並行する処理と捉えるのがわかりやすい．キャラクタ間の関係については，キャラクタ同士が情報をやりとりする通信と捉えることができる．

これらのことから，次の 4 つの処理が容易に記述できることを，ゲームシステムを記述するためのプログラミング言語の要件として挙げる．プログラミング言語はこれらの記述性に優れ，且つプログラミングに精通していないゲームデザイナーにも利用可能な簡潔な設計となっていることが求められる．

要件 1 キャラクタの状態遷移処理

要件 2 時間に沿った処理

要件 3 並行処理

要件 4 キャラクタ間の通信

この 4 項目の必要性はあくまで経験に基づいた仮説であるが、この 4 項目を前提とした検討を行い，研究によりゲームデザイナーによる実装とトライアンドエラーを容易にするという目的について一定の成果を得ることができれば，これら 4 項目を目標とすることの妥当性が間接的に示されることになると考える．

以下，本節では，これらの要件について，ビデオゲーム中での具体的な例を挙げて説明する．また，これらの機能を持たないプログラミング言語での記述から，これらの機能は，プログラミングに精通したプログラマーでなければ記述が難しいことと，ゲームデザイナーが利用できるようにするためには，言語によるサポートが必要な要件であることを説明する．

キャラクタの状態遷移処理　ゲーム中のキャラクタは,「歩いている状態」「ジャンプの状態」等の複数の「状態」と,「歩いている状態でボタンを押すとジャンプする」等の，状態の「遷移」で自然に表現できることが多い．状態と状態遷移は，ゲーム開発において，キャラクタの挙動を表現する方法としてよく知られている [Dyb00, Far04, Jac06].

これらの状態は，プレーヤーの操作やキャラクタ間の相互関係等から派生するキャラクタのアクションで，ゲームシステムのインタラクション性やゲームルールに強く関係する．移動と攻撃をするキャラクタを想定した簡単な例を**図 1.3** に示す.

この例では，キャラクタはプレーヤーの入力やアクションの終了により，次のシナリオで状態 (アクション) を切り替えている．

1. 登場時，キャラクタは「立ち」状態.
2. 移動ボタンで，キャラクタが画面中を走る (移動).
3. 移動ボタンで，キャラクタが「立ち」に戻る．
4. 攻撃ボタンで，キャラクタが「攻撃」をする．

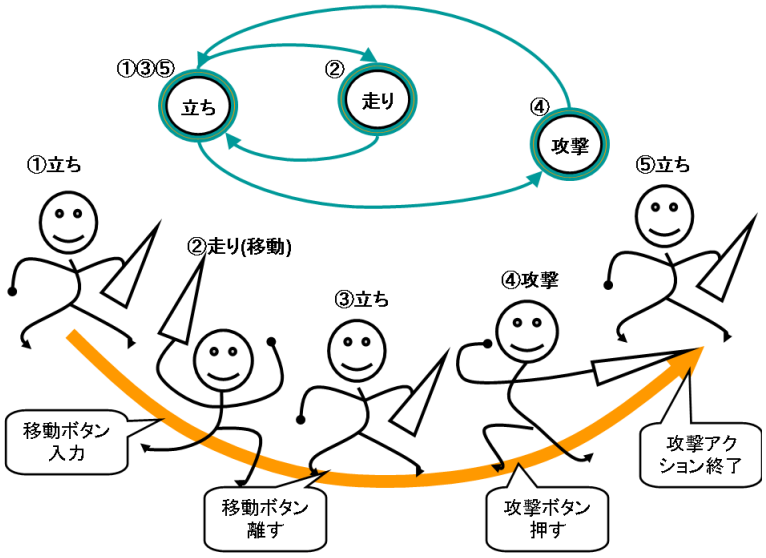

図 1.3: キャラクタの状態遷移の例

5. 攻撃が終了し，キャラクタが「立ち」に戻る．

キャラクタは「立ち」「走り」「攻撃」の 3 つのゲームルール上の状態を持ち，その間を遷移する．**図 1.3** の上部は，状態を円で，遷移を矢印で表した「状態遷移図」であるが，この図で表されるようなゲームキャラクタの挙動と変化を記述しやすい機能が，プログラミング言語には必要である．

状態遷移処理は，その記述のための特別な機能を持たないプログラミング言語を用いる場合，状態を記録する変数や分岐処理を用いて記述することができる．この例として**図 1.4** に，プログラミング言語 Java[Gos00] による記述を示す．このプログラムでは，ゲームキャラクタ `PlayerChar` の状態を 0 から 2 の整数で表し，`state` というキャラクタ固有の変数に記憶させ，キャラクタの処理中では，`switch` を使い `state` の値により，各状態処理へ分岐している．状態の遷移をする場合は，`state` 変数の値を目的の状態を表す整数に置き換えている．

状態遷移処理を記述する方法は**図 1.4** 以外にも，状態を手続きや関数として表現する方法 [Far04] や，オブジェクトとして表現する State パターン [GHV95] 等が知られているが，これらは拡張性や再利用性が**図 1.4** の方法より優れている一方で，記述はより複雑になる．

図 1.4 の記述方法は，複雑ではないが，キャラクタの種類や状態が増えたり，各状態で様々なゲーム上の処理を記述すると，変数や条件分岐の数が増え，状態の定義の見通しが悪くなり，状態を表す値や変数の定義や設定の間違いの原因となりやすく，プログラミングに慣れたプログラマーでも間違うことが少なくない．

```
class PlayerChar {
   static final int STATE_IDLE = 0;  // キャラクタ立ち状態
   static final int STATE_RUN = 1;   // 走り状態
   static final int STATE_PUNCH = 2; // 攻撃状態
   int state = STATE_IDLE;  // 初期状態
   // キャラクタの処理
   void update( ) {
     switch ( state ) {
     case STATE_IDLE: // 立ち状態処理
       ..
       state = STATE_RUN;  // 走り状態へ遷移
       ..
       break;
     case STATE_RUN:  // 走り状態処理
       ..
       state = STATE_IDLE;
       ..
       break;
     case STATE_PUNCH: // 攻撃状態処理
       break;
     }
   }
};
```

図 1.4: 状態遷移処理の記述サンプル

状態遷移処理はゲームキャラクタの挙動を自然に表現できることから頻繁に利用され，トライアンドエラーの過程で何度も修正が必要になる．間違いが入り込みやすく，見通しの悪い方法による記述は，トライアンドエラーの障害となる．プログラミングに精通していないゲームデザイナーのためのプログラミング言語には，この記述方法よりもわかりやすく，間違いの原因となりにくい機能が求められる．

時間に沿った処理　前述の状態遷移処理の他に，ゲーム中のキャラクタは，例えば「段々大きくなる」「10 メートルを 3 秒で移動して消える」「構えて 1 秒後に攻撃」等，ゲーム中の時間に沿った様々な状態の変化を伴う．ビデオゲームではこのような変化を表現するために 1/10〜1/60 秒単位でキャラクタの状態を変化させ画面を書き換えながらゲーム全体を進行させる．このアニメーションのコマに相当する時間の単位を「フレーム」と呼ぶ (**図 1.5**)．

リアルタイムなインタラクション性を有するゲームでは，ゲーム中の時間やフレームに合わせたゲームの展開が重要である．また，時間の経過によりゲームシーンが切り替わっていくような演出にも，ゲーム時間に合わせた処理の切り替えの実装が必要になる．

図 1.6 は，時間に沿った処理について特別な機能を持たないプログラミング言語の 1 つである Java 言語による，時間に沿った処理の記述例である．

図 1.6 では，キャラクタ `Char` に変数 `timer` をもたせ，毎フレームに 1 回呼び出される手続き `update` 内で，`timer` 変数を 1 フレーム時間だけ増加させることで，`timer` 変数に経過時間を記憶させている．`update` 内では `if-else`

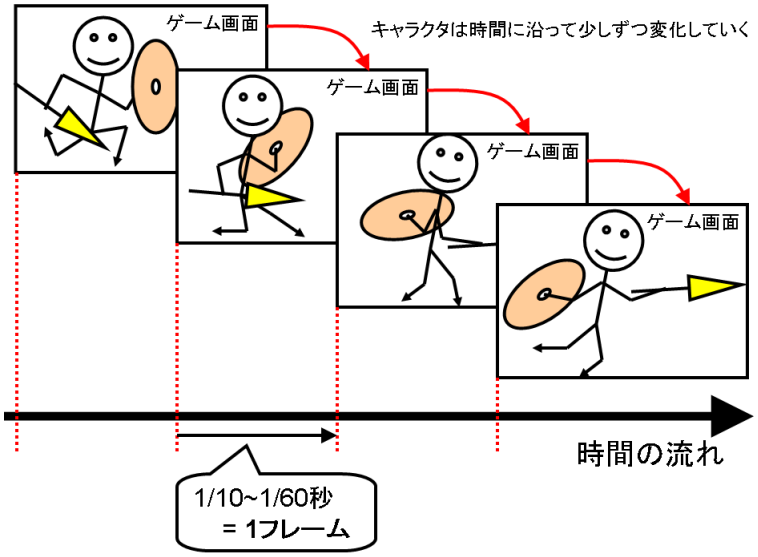

図 1.5: 時間に沿った処理とフレーム

を使って `timer` 変数の値で指定された時間に実行するべき処理へと分岐する．

この記述も難しいものではないが，前述の状態遷移処理と組み合わせて使用すると，状態遷移と時間に沿った処理のための 2 つの分岐処理が混在し，可読性が悪く，トライアンドエラーによる修正で間違いが混入しやすいものとなる．

可読性の悪い記述方法は，修正時に間違いが入り込む頻度が増え，頻繁なトライアンドエラーを困難にする要因となるので，プログラミングに精通していないゲームデザイナーの利用する方法として適切ではない．このような `timer` 変数や分岐処理を利用せずに，上から下への流れが直接時間の流れに対応し，処理を読解しやすい記述方法をプログラミング言語が提供すれば，時間に沿った処理の記述も容易になり，ゲームデザイナーによるトライアンドエラーをより容易なものにすることが可能になる．

並行処理　ビデオゲームでは複数のキャラクタが同時に登場し，それぞれが固有の状態遷移をおこないながら自律的に動作する (**図 1.7**)．また，1 つのキャラクタの処理に注目しても,「拳銃を相手に向けつつ一定間隔で撃ちながら走る」動作の実装は，複数の並行する処理として表現するのが自然である．

並行処理機能が無いプログラミング言語を使用する場合，キャラクタの処理を 1 フレーム時間分の処理を行う手続きとして記述し，その手続きを毎フレーム呼び出すことで自律した処理を実装する．この実装方法はゲームプログラミング技術として広く用いられている [Boe00, Har05]．

1 つのキャラクタに複数の並行処理を記述するには，各並行処理を 1 つの手続きとし，1 フレーム毎に必要な並行処理の手続きを全て呼び出すように

```
class Char {
   int timer = 0; // 時間変数
   // 毎フレームの処理
   void update( ) {
     if (timer == 30) { // 30 フレーム経過時の処理
        ...
     } else if (timer == 60) { // 60 フレーム経過時の処理
        ...
     } else if (timer >= 100) { // 100 フレーム以降の処理
        ...
     }
     timer += 1;   // 時間を進める
   }
};
```

図 1.6: 時間に沿った処理の記述例

する．**図 1.8** は，これを Java で記述した例である．この例では，キャラクタ PlayerChar に 2 つの並行する処理「歩く」と「撃つ」を記述している.「歩く」と「撃つ」はそれぞれ手続き walk と shoot として記述し，2 つの手続きを，キャラクタ全体の処理 update から呼び出している．個々の処理は独自の状態遷移処理や時間に沿った処理を持つので，その記述のために，状態を表す変数 walk_state，shoot_state や，時間の経過を表す変数 walk_timer，shoot_timer を用意している．update は 1 フレーム分の処理をする手続きであるので，呼び出される walk と shoot も 1 フレーム分の処理をする手続きとして記述している．

図 1.8 の記述方法は，walk や shoot のように並行する各処理が独自の状態遷移処理や時間に沿った処理を持つ場合，それぞれの手続きについて状態や経過時間を記録する変数の記述が必要となり，可読性が悪く，修正が容易ではないプログラムとなりやすい．また，どの部分が並行した処理として記述されているのかが明示できず，並行処理がどのような構成になっているのかも把握しづらい．これらが原因で，多数の自律した処理がある場合，記述は複雑になり，熟練したプログラマーであっても理解が容易ではなくなる．

並行処理は自律したキャラクタの挙動を記述する自然な方法であることから，できるだけ平易で，状態遷処理や時間に沿った処理の記述が複雑にならない記述方法を提供することで，ゲームデザイナーによるトライアンドエラーをより容易なものとすることが期待できる．

キャラクタ間の通信　ゲームシステムは，並行に動作するキャラクタが，お互いにゲームに関する情報を交換したり変更したりすることで成立する．例えば，プレーヤーキャラクタと敵キャラクタとの当たり判定処理や，その結果に対応したリアクション処理は，アクションゲームでは必須の処理である．このキャラクタ同士の競合性や，複数のキャラクタに関わるゲームルールの実装には，キャラクタ間の情報交換や共有処理が必要である．これらの情報交換や共有処理をまとめて，キャラクタ間の通信と呼ぶ．

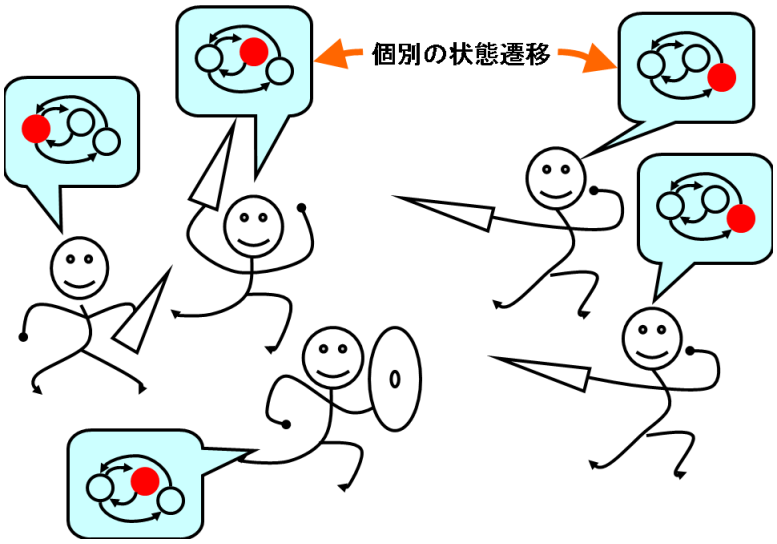

図 1.7: 状態遷移する複数のキャラクタによる同時並行処理

キャラクタ間の通信は，例えば「プレーヤーキャラクタは無敵状態中は敵キャラクタに体当たり可能」のように，キャラクタの種別だけでなく状態に依存したものであることも多い．従って，キャラクタの状態に依存したキャラクタ間の通信記述も容易である必要がある．

ゲームキャラクタ間には様々なルールや関係があるので，それらに適した記述機能の無いプログラミング言語では，通信方法に応じて異なる方法で記述しなければならない．そのため，現在，ゲームシステム記述に用いられているキャラクタ間の通信を表現するモデルも，手続き呼び出し，コルーチン，イベント駆動型プログラミング，グローバル変数と，複数ある．これらについては，2.6.1 節で詳しく述べる．

例えば，あるキャラクタ (親) にもう一つのキャラクタ (子供) が常についてくる挙動を記述する場合，Java では，**図 1.9** のような記述をする．この例には，2 つのキャラクタ `ParentChar`(親キャラクタ) と `ChildChar`(子供キャラクタ) が記述されている．2 つのキャラクタとも毎フレーム `update` が呼び出されて，1 フレーム分づつ処理が進められる．親キャラクタの毎フレームの処理 `update` からは，子供キャラクタの位置を設定する `set_position` が呼び出されており，これにより，子供キャラクタが親キャラクタについてくるようになっている．

この記述は，`ParentChar` キャラクタと `ChildChar` キャラクタ間の子供がついてくる，という通信のみを処理するものである．そのため，この 2 つのキャラクタ間で他の種類の通信が必要な場合は，`set_position` とは別の記述が必要になり，通信として明示的にわかりやすく記述することを難しくする．

また，通信は，親キャラクタと子供キャラクタの両方に必要で，子供キャ

```
class PlayerChar {
   int walk_state = ...;  // 歩く処理の状態を表す変数
   int walk_timer = 0;    // 歩く処理用のタイマー
   int shoot_state = ...; // 弾を撃つ処理の状態を表す変数
   int shoot_timer = 0;   // 弾を撃つ処理用のタイマー
   // 毎フレームの処理
   void update( ) {
     walk( );  // 歩く処理の呼び出し
     shoot( ); // 弾を撃つ処理の呼び出し
   }
   void walk( ) {  // 1フレーム分の歩く処理
     .. 状態遷移と時間に沿った処理 ..
   }
   void shoot( ) { // 1フレーム分の弾を撃つ処理
     .. 状態遷移と時間に沿った処理 ..
   }
};
```

図 1.8: 並行処理の記述例

ラクタ側の set_position は，子供キャラクタの本来の処理 (update) とは別に記述しなければならないことから，通信がキャラクタの状態に依存するような場合，set_position でもキャラクタの状態を検査することが必要となる．このため，通信に関する記述が 2 つのキャラクタに分散し可読性も悪くなる．

子供キャラクタが複数あり，全子供キャラクタに対して同時に通信を行う場合は，子供キャラクタをまとめて管理する**図 1.10** のような記述が必要となる．この記述では，親キャラクタに children という，関連する子供キャラクタ全てを記録する変数を用意し，毎フレームこの children に記録されている子供キャラクタ全てについて set_position を呼び出すことで，全ての子供キャラクタが親キャラクタについてくるようになっている．**図 1.9** との処理の違いは子供が複数になっただけであるが，記述においては，複数キャラクタを管理するコードへの修正が必要となっている．

プレーヤーキャラクタと敵キャラクタが衝突したときに，自分にポイントを加算して，相手にダメージを与えリアクションさせるような処理を記述できるようにするために，衝突時に衝突したキャラクタの特定の手続きが呼び出されるような規約を導入し，**図 1.11** のように記述することが多い．

この記述では，キャラクタが衝突した場合，衝突したキャラクタの collides という手続きが，もう一方の衝突相手のキャラクタを引数に呼び出される規約となっている．PlayerChar は引数が敵キャラクタ (Enemy) であるかどうかを検査して，敵キャラクタであるなら，ポイント計算やリアクションの処理を行なっている．

この場合，PlayerChar に対するあらゆるキャラクタとの衝突時に collides が呼ばれるために，ゲームルール上，処理が必要な衝突かどうかを個別の collides 内で記述しなければならない．さらに，キャラクタの状態に依存して衝突処理が変わる場合，状態に応じた分岐処理も必要となる．衝突に関す

```
class ParentChar {  // 親キャラクタ
   int x, y;    // 親の位置
   ChildChar child = ...; // 子供キャラクタ
   void update( ) { // 毎フレームの処理
     ..
     child.set_position(x, y); // 子供キャラクタの位置設定
     ..
   }
};
class ChildChar { // 子供キャラクタ
   int x, y;  // 子供の位置
   void update( ) { // 毎フレームの処理
     ..
     child.set_position(x, y); // 子供キャラクタの位置設定
     ..
   }
   void set_position( int _x, int _y ) { // 位置の設定
     x = _x; y = _y;
   }
}
```

図 1.9: 親子関係にあるキャラクタ間の通信の記述例

```
class ParentChar {  // 親キャラクタ
   int x, y;    // 親の位置
   ChildChar children[] = ...; // 子供キャラクタの列
   void update( ) { // 毎フレームの処理
     ..
     // 全ての子供キャラクタについて処理
     for ( ChildChar c : children ) {
        c.set_position(x, y); // 子供キャラクタの位置設定
     }
     ..
   }
};
```

図 1.10: 多数の子供キャラクタに同報的に通信する記述例

るゲームルールが単純であればこの記述方法でも十分ではあるが，多種類のキャラクタについてキャラクタの状態に応じて衝突処理が異なる場合，複雑な条件分岐が必要となり，記述の修正も難しくなる．

また，個々のキャラクタに `collides` 手続きを記述しなければならず，衝突処理をキャラクタの処理とは明示的に分けて記述できないので，プログラムが大きくなると多くのキャラクタ定義中に衝突処理が記述され，衝突処理はコード中に分散する．このため，`PlayerChar` と `Enemy` の衝突処理をどちらのキャラクタにも記述してしまうような重複記述の誤りや，どのキャラクタに衝突処理が記述されているのかがわかりづらくなる可読性の問題があり，記述の誤りの原因となりやすい．

可読性が悪く，誤りの混入しやすい記述方法は，ゲームデザイナーの修正と検証を難しくし，トライアンドエラーによる開発を妨げる原因となる．キャラクタ間の通信は，キャラクタ同士の競合性や，複数のキャラクタに関わる

```
class PlayerChar {
   int point = 0;
   // PlayerChar に何かが衝突した際に呼び出される.
   void collides( Char other ) { // other は衝突したキャラクタ
     if ( other instanceof Enemy ) { // 衝突した相手が敵キャラの場合
       point += 10; // ポイント加算
       other.damage( 10 ); // ダメージを与える
       other.reaction( DAMAGE_REACTION ); // リアクション設定
     }
   }
};
```

図 1.11: キャラクタの衝突によるポイント計算やリアクション処理の例

ゲームルールの実装に不可欠であることから，頻繁なテストや修正を必要とする．トライアンドエラーを何度も繰り返すゲームデザイナーの作業を円滑にするために，プログラミング言語は，キャラクタ間の通信をわかりやすく，修正も容易な記述方法を提供することが求められる．

1.10 本研究の目的と方針

2 章で詳しく述べるが，現在ビデオゲーム開発において用いられている記述ツールには，ゲームデザイナーがトライアンドエラーを何度も繰り返すための記述ツールとしては，次の問題点がある．

- ゲームシステムを記述するのに適切な機能を十分に提供していない．
- ゲームデザイナーがゲームシステムの全てを記述できない．

多くの記述ツールがゲームシステムを記述するのに十分な機能を提供していない．ビデオゲーム開発を目的としたプログラミング言語には，ゲームシステム記述に適した機能を提供するものもあるが 1.9 節の 4 つの要件を満たすものは無く，十分とは言えない．統合的なビデオゲーム開発環境は，容易にゲームアプリケーションの開発が可能であるが，グラフィカルインターフェースだけでは，ゲームシステムの記述には不十分で，提供されているプログラミング言語もゲームシステムの記述に求められる 4 つの要件を満たしていない．個々のゲーム専用の記述ツールは，特定のゲームシステム記述に特化した記述方法であるため，他のタイプのゲームシステムの記述に適しておらず，類似するその他のツールは，ビデオゲームを目的としていないので，複雑なゲームシステム記述に適用することが難しい．

また，ゲームシステムを記述するのに機能が十分でないことから，ゲームデザイナーだけではゲームシステムの実装と検証を十分に行うことができない．そのため，不足する機能についてはプログラマーの協力が必要となり，トライアンドエラーを繰り返すプロセスの障害となる．

ゲームデザイナーがプログラマーの協力無しにゲームシステムを記述し，実装とテストを直接行うことができれば，トライアンドエラーを繰り返すこ

とが容易となる．ゲームデザイナーがゲームシステム全てを記述可能とするためには，1.9 節の 4 つの要件を満たす新しいプログラミング言語が必要であり，これら 4 つの要件を満たすプログラミング言語の設計と開発を行い，それをビデオゲーム開発で使用することが，現在の開発をより円滑に進めるために有効な方法と考えられる．

本研究では，1.9 節で述べた要件を満足するような，新しいゲームシステム記述用のプログラミング言語を設計及び開発することを目的とする．そのために，既存の記述ツールやプログラミング言語やシステムの調査を行った上で設計と実装をおこなう．さらに評価のため，開発したプログラミング言語を用いて，ビデオゲーム開発も行う．

1.11　まとめと本論文の構成

本章では，ビデオゲーム業界と開発の現状について説明し,「面白さ」という定義や事前の検証が難しい価値を提供しなければならないビデオゲームのためには，トライアンドエラーによる開発が不可避であることを説明した．さらに，開発中の製品イメージの共有やトライアンドエラーのためにゲームシステム記述ツールが用いられており，それをゲームデザイナーが直接使用することで，ゲームデザイナーとプログラマーの共同作業を軽減するという形で，ビデオゲーム開発で有効に利用されていることを説明した．

また，ゲームシステム記述ツールには様々なものが利用されていることも説明した．これらの記述ツールは，ビデオゲーム開発で有効に利用されているものの，ゲームシステムをゲームデザイナーだけで記述するには機能不足であるため，プログラマーの協力が必要となっており，トライアンドエラーを阻害する要因となっていることを述べた．

この解決のためには，ゲームデザイナーがゲームシステムを記述するのに十分な機能を持つプログラミング言語を提供することが有効な方法であり，そのプログラミング言語は「状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」の 4 つの要素の記述が容易な機能が必要であることを述べ，本論文では，これら 4 つの要件を満たすプログラミング言語の設計と開発を目的とすることを説明した．

以下，2 章では，調査した既存の記述ツールやゲームシステムの記述に関連した研究について詳細に説明する．3 章では，本研究において設計開発した新しいプログラミング言語 S540 の設計と実装について説明し，その適用結果と評価を報告する．4 章では，3 章で開発した S540 の評価からわかった問題点を解決するための新しいプログラミング言語機構 Join Token の提案と，とそれを組み入れたプログラミング言語 Mogemoge について説明する．5 章では，本論文を総括し，結論を記す．

第2章　関連研究

2.1　はじめに

1 章では，ゲームデザイナーが直接利用可能なゲームシステム記述ツールは，ゲームデザイナーとプログラマーの間のやり取りを軽減し，ビデオゲーム開発におけるトライアンドエラーを容易にすることを述べた．

ゲームシステム記述ツールは，ゲームシステムを記述するのに十分な機能のためにプログラミング言語を提供し，そのプログラミング言語は「状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」の 4 つの要素の記述を容易にする機能を持つ必要があることも述べた．

現在ビデオゲーム開発に利用されている記述ツールには，商用，非商用を問わず様々なものが存在する．また，ビデオゲームを目的としていないが，それに近いアプリケーション開発を目的としたツールも多く存在する．

本章では，これら既存の記述ツールを調査し，1 章で述べた記述ツールに求められる 4 つの要件をどの程度満たしているかについて検討を行う．調査したツールは大きく次の 4 つに分類した．

- ビデオゲーム開発を目的としたプログラミング言語
- 統合的なビデオゲーム開発環境
- 個々のゲームの開発専用に作られた記述ツール
- 類似するその他のツール

ビデオゲーム開発を目的としたプログラミング言語とは，ゲームアプリケーションに組み込まれて利用されることを目的としたプログラミング言語やその処理系である．

統合的なビデオゲーム開発環境とは，ビジュアルインターフェースを用意し，グラフィカルに開発を進められ，グラフィクスやサウンド等のデータの利用も容易な，ゲームアプリケーションの開発環境を指す．

個々のゲームの開発専用に作られた記述ツールは，開発目的とする製品の開発に特化した記法や機能を提供するゲームシステム記述ツールである．記述ツールを特定のゲーム専用に特化した設計や実装にすることで、目的とするゲームシステム記述に関して優れた機能を提供し，開発するゲームプログラムに適した実装とすることができる。

類似するその他のツールは，ビデオゲームに類似したアプリケーションの制作が可能なものを指す．プログラミング教育用のツールや，エージェントシミュレーション，アニメーションオーサリングツール等がある．

また，調査と検討の結果，既存の記述ツールには 1 章の要件のうち特に「キャラクタ間の通信」について，考慮されていないか，限定的な範囲の通信を記述する機能しかもたないものが大部分であった．さらに，ビデオゲームのキャラクタは様々な通信を行うので，キャラクタ間の通信を記述するための機能としてどのようなものが有効であるかはっきりしていない．既存の記述ツールにはキャラクタ間の通信のための十分な機能を提供しているものがないため、既存のツールの機能に基づいて通信機能の検討を行うことは難しい

ビデオゲームのキャラクタが自律的に処理を進めながら通信を行いゲームを進める処理は，プロセスやスレッド等の並行処理と，その並行処理間で通信を行うことで処理を進める計算機プログラムに合致している．そこで，計算機科学分野で研究されている並列/並行処理間の通信モデルについて調査し，キャラクタ間の通信のモデルとして適切かどうかの検討も行う．

以下，本章では，2.2 節で「ビデオゲーム開発を目的としたプログラミング言語」の調査結果について，2.3 節で「統合的なビデオゲーム開発環境」の調査結果について，2.4 節で「個々のゲームの開発専用に作られた記述ツール」の調査結果について，2.5 節で「類似するその他のツール」の調査結果について述べる．その後，2.6 節で，通信モデルについての調査について説明し，2.7 節で本章の調査と評価について総括する．

2.2　ビデオゲーム開発を目的としたプログラミング言語

2.2.1　ビデオゲーム開発を目的としたプログラミング言語の位置づけ

ビデオゲーム開発の記述ツールとして用いられるもの中には，ビデオゲーム開発に利用されることを目的としたプログラミング言語がある．また，元はビデオゲーム記述が目的ではないが，ビデオゲーム開発の記述ツールとして利用するために修正や拡張をして利用されているプログラミング言語もある．

これらのプログラミング言語は，新しくプログラミング言語の設計や実装をすることが簡単ではなく，既に実装として存在するものを利用する方が容易で開発の手間や時間を節約できることと，プログラミング言語の学習のために必要とするドキュメントや情報が用意されていることから，広く利用されている．

調査は，ビデオゲームアプリケーション開発での利用実績があるものについて行った．本節では，調査したプログラミング言語について，機能や特徴について説明し，1.9 節の 4 つの要件についての評価を行う．また，最後の 2.2.9 節では，各言語の評価をまとめる．

2.2.2 Lua

Lua[IdFF96] はアプリケーションにカスタマイズ機能を提供するためのスクリプト言語として，アプリケーション組み込み用に開発されたプログラミング言語および言語処理系である．特にリアルタイムアプリケーションへの適用を考慮し，実行時の効率性や組み込みの容易さ等を目標として設計/実装されている．ゲームアプリケーションへの適用例も少なくない [Emm09, Gro10].

Lua は次のような様々な組み込み方が可能である．

- ゲームの毎フレームのメインルーチンとして Lua スクリプトを実行する．
- 衝突等のゲーム中のイベント処理時に，指定名の Lua 関数をキャラクタを表すデータを引数として呼び出す．
- ゲームやキャラクタの初期化時だけ特定のスクリプトを実行する．

Lua は次の特徴を持つ．

- C 言語や Perl[WCO02] のような手続き型の言語である．
- 連想配列型を利用して，オブジェクト指向プログラミングが可能である．
- 並行する処理の記述に便利なコルーチンがある．
- 組み込み時の効率性を考慮した設計や実装となっている．

Lua は，C 言語や Perl と同じく 1 つの処理を手続きとして定義し，それらの組み合わせによりプログラムを記述する動的型付けのプログラミング言語である．

Lua では，組み込みデータ型である連想配列を，Self[US87],JavaScript[Fla07], ドリトル [兼宗 01] 等のプロトタイプ方式の言語のようにオブジェクトとして利用可能である．ゲーム中のキャラクタを 1 つのオブジェクトとして定義し，オブジェクトに属する手続きを衝突等の処理記述用途に利用することも可能である [VR02].

また Lua は，並行する処理を協調的に記述するのに便利なコルーチンや，コルーチンを元にした並行処理を記述するのに便利なライブラリを提供する．

Lua の実装は，ビデオゲーム等のリアルタイムアプリケーションへの適用を考慮し，組み込みが容易で，実行時効率性を重視したものとなっている [IdFC05]．また，アプリケーション固有の機能をサポートさせるために，組み込んだアプリケーション専用のライブラリや命令を追加することが容易な実装になっていることも特徴として挙げられる．実行時にネイティブコードを生成することで，高速な処理を可能にする JIT コンパイラ [Pal05] もある．

図 2.1 は Lua の記述例である．これは，Lua のオブジェクトにゲームキャラクタを対応させて記述する形式で Lua を組み込んだ場合の一例であり，Lua を組み込むと必ずこのような記述になるわけではない．この例では，`Char` という連想配列をキャラクタのベースとなるオブジェクトと定義し，`Char.new`

```
-- キャラクタの定義
Char = {}
Char.new = function(_x,_y,_vx,_vy)
  local c = {}
  c.position = { x:_x, y:_y }   -- キャラクタ位置
  c.velocity = { x:_vx, y:_vy } -- キャラクタ速度
  c.state = "idle"
  -- 毎フレームの処理.
  c.update = function()
    if    self.state == "idle" then idle()
    elsif self.state == "run"  then run()
    end
  end
  -- 立ち状態の 1 フレーム分の処理
  c.idle = function()
    if onPadButton() then state = "run"; end
  end
  -- 走り状態の 1 フレーム分の処理
  c.run = function()
    if !onPadButton() then state = "idle"; end
  end
end
-- 実行時に毎フレーム呼ばれる関数 (組み込み側プログラムで指定)
function handleFrame()
  for i = 1, CHAR_COUNT do
    c = CharArray[i]
    c.update()
  end
end
-- キャラクタを 2 つ生成
CharArray = {}
CharArray[0] = Char.new(10, 20, 1, 1)
CharArray[1] = Char.new(40, 100, 0.5, 3)
CharArray[2] = Char.new(40, 0, 2, -2)
```

図 2.1: Lua のサンプル

関数でキャラクタの生成，Char:update 関数でキャラクタの毎フレームの動作を記述している．

組み込んだアプリケーション側は必要な処理が記述されている手続きを指定してプログラムの実行を進める．例えば，**図 2.1** では，handleFrame 関数が毎フレーム実行されるように組み込むことで，定義された全てのキャラクタの update 関数が呼び出されるようになっている．

Lua は 1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 並行処理を記述する機能がコルーチンしかない．

要件 4 キャラクタ間の通信の記述に適した機能が無い．

Lua には，状態遷移処理の記述に適した機能が無いため，Lua で状態遷移処理を記述するには，1.9 節の**図 1.4** のように，状態を表す変数と条件分岐を用いて記述しなければならず，見通しの悪さや間違いの原因となりやすい．

また，Lua には時間に沿った処理を記述する機能もない．よって，**図 1.6** のように時間を保持する変数と分岐処理により記述しなければならず，これも誤りの原因となりやすい．

Lua では，コルーチンを使うことで，お互いに協調しながら処理を進めるプログラムの記述が比較的容易である．しかし，コルーチンは，コルーチンとして呼び出される側が処理を中断して呼び出し元へ処理を戻さなければ他の並行するコルーチンとの連携ができず，生成や消滅を繰り返しながら自律的に処理を進めるゲームキャラクタや，キャラクタ間の通信の記述には適していない．コルーチンについては，2.6.3 節でも議論する．

Lua には自律して動作するゲームキャラクタ間の様々な通信の記述に適した方法が無い．このため，通信は処理内容に応じて様々な記述方法を使い分けなければならず，キャラクタの通信をわかりやすく明示することも難しい．

以上から，Lua は 1.9 節の 4 つの要件全てを満足していない．

2.2.3 Pawn

Pawn[Rie99] [1] は，アクションゲーム「Grand Theft Auto: San Andreas」[Roc04] のネットワークマルチプレーヤー版において，サーバーサイドスクリプト言語として利用されたプログラミング言語である．サーバーサイドをスクリプト言語で記述できるようにすることで，Grand Theft Auto の運営者が，ゲーム難易度の変更をしたり，独自ルールのゲームを提供することが容易になっている．

Pawn も Lua と同じく，アプリケーションに依存しない，汎用の組み込み用プログラミング言語として開発された言語であり，ゲーム以外にも様々なアプリケーションにおいて組み込み言語として利用されている．

Pawn は次の特徴を持つ．

- C に類似した構文の手続き型の言語である．
- キャラクタの状態遷移を記述するのに適した State という機能と，ゲーム中のイベント処理に適した Event という機能がある．
- C/C++プログラムへの組み込みを意識した実装である．

Pawn は手続き型で動的型付けの言語であり，プログラムは関数の集まりとして記述する．オブジェクト指向言語ではなく，オブジェクトやクラスという概念は存在しない．

[1] Pawn は Small という名前であったが呼称を変更している．`http://www.compuphase.com/small.htm`

```
/* キャラクタの処理の記述 */
char_main(ch) {
  state idle  /* 最初の状態:立ち状態 */

  @keypresssed(key) <idle> { state(isPadButton(key)) run }
  @keypresssed(key) <> { }  /* fallback */

  @keyreleased(key) <run>  { state(isPadButton(key)) idle }
  @keyreleased(key) <> { }  /* fallback */

  entry() <idle> { /* 立ち状態に入ったときの処理 */ }
  entry() <run>  { /* 走り状態に入ったときの処理 */ }
}
```

図 2.2: Pawn のサンプル

Pawn はゲームシステムの記述に適した機能として，キャラクタの状態遷移を記述するのに適した State という機能と，ゲーム中のイベント処理に適した Event という機能がある．State は状態を表す識別子を関数やイベントに関連付けるもので，Event はユーザーの入力等のイベントに応じた処理を記述する機能である．2 つを組み合わせることで，イベントに応じた状態遷移処理の記述ができる．

また，Lua 同様，主に組み込みを目的としているので，軽量で組み込みがし易い設計と実装がなされている．

図 2.2 は Pawn による記述例である．プログラムは `char_main()` や `entry()` のような関数の集まりとして記述される．この例では，Pawn の State と Event 機能を用いて，立ち状態と走り状態の二つの状態を持つキャラクタの動作を記述している．イベント処理は次のように記述する．

`@`イベント名 `<`状態名`>` `{` 状態遷移処理の記述 `}`

例えば，**図 2.2** では，`idle` 状態でパッドのボタン押下イベントが発生したら，`run` 状態へ遷移する．遷移する条件と遷移先は `state` 文で記述する．各状態へ遷移したときは特別な関数 `entry()` が呼ばれる．初期状態は関数の最初の `state idle` として記述されている．**図 2.1** の例と比較すると，状態遷移が分かりやすく記述可能となっている．

Pawn は 1.9 節の 4 つの要件について，次のように評価できる．

要件 1 State や Event 機能で状態遷移処理の記述が容易である．

要件 2 時間に沿った処理の記述に適した機能は無い．

要件 3 並行処理を記述する機能はない．

要件 4 キャラクタ間の通信を記述する機能として Event が利用できるが，限定された場合にしか利用できない．

Pawn にはキャラクタの状態を記述するための State と，ゲーム中のイベントに関連した遷移の記述に適した Event があるので，状態遷移処理をわかり

やすく記述することが可能である．状態を表す変数は必要なく，条件分岐処理などの記述を減らすことができる．

Pawn には，時間に沿った処理の記述に適した機能は無い．そのため，1.9 節の**図 1.6** のように，時間を保持する変数と分岐処理で記述しなければならない．

Pawn には並行処理を記述する機能もない．並行処理を記述するには，1.9 節の**図 1.8** のように，1 フレームづつ処理を進める手続きを複数記述し，毎フレーム呼び出すような記述をしなければならない．

Pawn には Event があり，指定したイベントをきっかけに状態遷移を行う処理が記述可能である．イベントは，キャラクタ間通信の手段としても利用することはできるが，通信相手は必ず 1 つの状態遷移処理を行うプログラムであり，複数のキャラクタに対して一度に通信を行ったり，衝突処理のような複数のキャラクタが一度に関係する処理の記述には適していない．

以上から，Pawn は，状態遷移処理については十分な記述機能を持つが，その他については 1.9 節の 4 つの要件を，十分に満足しているとは言えない．

2.2.4 Squirrel

Squirrel[Dem04] も前述の Lua や Pawn と同じく，アプリケーションに依存しない，汎用の組み込み用プログラミング言語として設計及び実装された言語である．

Squirrel は，ロールプレイングゲームの「小さな王様と約束の国ファイナルファンタジー・クリスタルクロニクル」[ffc08] の開発用の組み込み言語として使用された．小さな王様と約束の国は，プログラムの 7 割程度が Squirrel で記述されたと報告されている [白石 08]．

Squirrel は次の特徴を持つ．

- Python や Ruby 言語に似た，クラスベースの手続き型のオブジェクト指向言語である．
- 協調する並行処理の記述に利用できるコルーチンを持つ．
- C/C++プログラムへの組み込みを意識した実装である．

Squirrel は Python や Ruby 言語に似た，クラスをベースにした手続き型オブジェクト指向言語である．前述の Lua や Pawn と違い，クラスとそのメソッドをプログラムの構成単位としている．ゲームキャラクタをクラスのインスタンスとして記述可能となるよう組み込むことで，ゲームキャラクタの記述や実装が Lua や Pawn よりも容易になると考えられる．

Squirrel も Pawn と同じく，コルーチンの機能を持つ (Squirrel ではスレッドという名前で呼ばれる)．小さな処理を複数協調動作させるような処理の記述などに利用可能である．

```
// キャラクタの定義
class Char {
  position = {}; velocity = {};
  state = "idle";
  constructor(_x,_y,_vx,_vy) {
    // キャラクタ位置と速度の設定
    position.x = _x;  position.y = _y;
    velocity.x = _vx; velocity.y = _vy;
  }
  // 毎フレームの処理
  function update() {
    if      state == "idle" { idle(); }
    else if state == "run"  { run(); }
  }
  // 立ち状態
  function idle() {
    if ( onPadButton() ) { state = "run"; }
  }
  // 走り状態
  function run() {
    if ( !onPadButton() ) { state = "idle"; }
  }
}
// キャラクタを 3 つ生成
CharArray = []
CharArray.push( Char(10, 20, 1, 1) )
CharArray.push( Char(40, 100, 0.5, 3) )
CharArray.push( Char(40, 0, 2, -2) )
// 実行時に毎フレーム呼ばれる関数 (組み込み側プログラムで指定)
function handleFrame()
  CharArray.map( function(c) { c.update(); } )
end
```

図 2.3: Squirrel のサンプル

また，Lua や Pawn と同様，組み込みを目的とした言語であるので，アプリケーションへの組み込みが容易となるような設計及び実装となっている．

前述の Lua のサンプル (**図 2.1**) を Squirrel のプログラムとして書いたものが，**図 2.3** である．Lua のサンプルでは連想配列をキャラクタに対応させているところをクラスとして記述できるため，**図 2.1** の「`c.`」のような修飾や `new` 関数の中で他の関数を記述するような必要はなく，わかりやすく記述できている．

Squirrel も Lua や Pawn と同様に組み込みが容易であることと,「小さな王様と約束の国ファイナルファンタジー・クリスタルクロニクル」のような大きなゲーム開発プロジェクトで採用され，使用報告 ([白石 08]) もあったことから，日本での知名度は高い．

Squirrel は 1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能は無い．

要件 2 時間に沿った処理の記述に適した機能は無い．

要件 3 並行処理を記述する機能がコルーチンしかない.

要件 4 キャラクタ間の通信の記述に適した機能が無い.

Squirrel には，状態遷移処理の記述に適した機能が無い．そのため，**図 1.4** のように，状態を表す変数と条件分岐を用いて記述しなければならない.

Squirrel には時間に沿った処理を記述する機能もない．時間に沿った処理を記述するには**図 1.6** のような記述方法を用いなければならず，見通しの悪さや誤りの原因となりやすい.

Squirrel にはコルーチンがあるが，定型的な並行処理の記述には適していても，自律的に処理を進めるゲームキャラクタの処理の記述には適していない.

Squirrel にはキャラクタ間の通信を記述するのに適した機能もないので，通信を明示しわかりやすく記述することが難しい.

以上から，Squirrel は 1.9 節の 4 つの要件全てについて十分に満足しているとは言えない.

2.2.5 JavaScript

JavaScript[Fla07] は，Web ブラウザ上におけるページの表示や動作を記述することを目的とした，プログラミング言語である．近年では，ECMAScript [ECM99] として標準化されつつある.

さらに近年では，解説書などが多く出版され学習が容易なことや，オブジェクト指向言語でゲーム中のキャラクタの振る舞いの記述に適していることから，いくつかのゲーム開発環境用の記述言語としても採用されている [Tec06, CHFL11]．また，CRISCRIPT[松田 09] のように可搬性のある組み込み用の JavaScript 実装も存在する.

JavaScript は，Self[US87], ドリトル [兼宗 01] 等と同じプロトタイプ方式のオブジェクト指向言語である．ゲームキャラクタをオブジェクトに対応させ，ゲームキャラクタの動作をそのメソッドとして記述することが可能である.

HTML5[W3C11] には canvas と呼ばれる新しいグラフィクス描画要素が追加されており，これと JavaScript を使用することで，HTML ドキュメント単体でリアルタイムゲームアプリケーションを記述することが可能である．**図 2.4** は，HTML5 のサンプルコードである．`Char` オブジェクトはゲームキャラクタの記述で，`update` や `draw` が毎フレーム呼ばれるようにして，リアルタイムアプリケーションの基礎部分を記述している.

JavaScript は 1.9 節の 4 つの要件について，次のように評価できる.

要件 1 状態遷移処理の記述に適した機能が無い.

要件 2 時間に沿った処理の記述に適した機能が無い.

要件 3 並行処理を記述するのに，コルーチンはあるが十分ではない.

要件 4 キャラクタ間の通信の記述に適した機能が無い.

```
<html><head>
<script>
  canvas = document.getElementById( "myCanvas" );
  context = canvas.getContext( "2d" );
  setInterval( update, 25 ); // 25ms おきに update 呼び出し
  //キャラクタオブジェクト
  var Char = function(_x,_y,_vx,_vy) {
     var o = {}
     o.x = _x; o.y = _y;
     o.vx = _vx; o.vy = _vy;
     o.update = function() { // 毎フレームの処理
       o.x += o.vx; o.y += o.vy;
     };
     o.draw = function() { /* context を用いて描画の処理 */ };
     return o;
  }
  //全キャラクタオブジェクトの処理
  function update() {
    context.clearRect(0, 0, canvas.width, canvas.height);
    for ( var n = 0;  n < objs.length;  ++n ) {
      objs[n].update(); objs[n].draw();
    }
  }
  // キャラクタの生成
  objs = new Array();
  objs.push( Char( 10, 50, 1, 1 ) );
  objs.push( Char( 50, 100, 2, -2 ) );
</script><body>
<canvas id="myCanvas" width="200" height="200"></canvas>
</body></html>
```

図 2.4: HTML5(JavaScript) を利用したサンプル

JavaScript はブラウザ上のアプリケーション開発用に発展してきたプログラミング言語で，ゲームアプリケーションへの組み込みやゲームシステム記述への適用を考慮されていないため，ゲームシステム記述に適した機能を持たない．そのため，JavaScript は 1.9 節の 4 つの要件全てを満たしていない．

2.2.6 Stackless Python

Python[Lut98] は，汎用のオブジェクト指向言語で，ビデオゲーム開発を目的としてはいない．しかし，Python には組み込み用として言語と実装に修正を加えた Stackless Python[Tis00] があり，いくつかのビデオゲーム開発用のスクリプト言語として用いられている [Daw02a, syl07].

Stackless Python は，元の Python と次の点で異なっている．

- 並行処理を記述するのに利用できる，コルーチン，軽量スレッド，タスクレットの機能を持つ．
- 並行処理間の通信を記述するのに便利なチャネル機能を持つ．

コルーチンや軽量スレッドは，ゲーム中のキャラクタの並行処理を記述するのに有効である．タスクレットはコルーチンを使いやすくしたもので，並行する処理をタスクレットとして記述することで，コルーチンよりも自由に並行する処理の生成やスケジューリングが可能である．

タスクレット同士の通信に便利なチャネルという仕組みも用意されている．チャネルは双方向の情報のやりとりに利用出来るオブジェクトで，イベントやコマンドの待ち受け等に利用できる．

次は，Stackless Python のタスクレットを用いた記述例である．3 つのタスクレットが順番に 2 回出力を行う．

```
def mytask(x):
    print "1:", x
    stackless.schedule()  # 他のタスクレットの処理へ
    print "2:", x
    stackless.schedule()  # 他のタスクレットの処理へ
end
stackless.tasklet(mytask)('first')
stackless.tasklet(mytask)('second')
stackless.tasklet(mytask)('third')
```

stackless.schedule() は，処理を中断し，他のタスクレットへ処理を移譲する命令である．stackless.schedule() を使うことで，複数のタスクレットが 1 フレーム分づつ処理を進める記述が可能である．このプログラムの結果は次のようになる．

```
1: first
2: first
1: second
2: second
1: third
2: third
```

stackless.schedule() 命令を用いたタスクレットの記述は，キャラクタ全てが平等に 1 フレームづつ処理を進めるビデオゲームの仕組みに近い．そのため，タスクレットを用いてビデオゲームの仕組みを反映した記述ができると考えられる．

次は簡単なチャネルの例である．

```
ch = stackless.channel()
def recv():
    ch.receive()
    print "受信"
def send():
    ch.send("こんにちは")
```

1 つのチャネルを複数の処理で共有して情報を送受信することで通信の記述ができる．チャネルは stackless.channel() で生成する．チャネルに対して send() メソッドを使うことで，チャネルを通してメッセージを送信し，同

じチャネルに対して receive() メソッドを使うことで，送信されたメッセージを受信できる．receive() はチャネルを通してメッセージを受信できるまで処理を停止するので，ゲームシーン中のイベントを待ち受ける処理の記述に使用できる．

タスクレット間でチャネルを共有すれば，タスクレット同士の通信を記述することが可能である．Stackless Python では，タスクレットとチャネルを用いることで，自律的に処理を進めながら相互に通信し合うプログラムの記述が容易となっている．

Stackless Python はゲームシステム記述に適した機能を持つが，1.9 節の 4 つの要件全てを満足するわけではない．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 タスクレットと schedule 命令を組み合わせることで，時間に沿った処理をある程度容易に記述できる．

要件 3 コルーチンやタスクレットで並行処理の記述は可能だが，十分ではない．

要件 4 キャラクタ間の通信の記述にチャネル機能を利用できるが，利用できる場合が限られる．

Stackless Python には，状態遷移処理の記述に適した機能が無い．状態遷移処理は，状態を表す変数や条件分岐を用いて記述しなければならず，見通しの悪さや誤りの原因となりやすい．

Stackless Python には，タスクレットの schedule 命令を用いて，1 フレーム時間分だけ処理を停止する記述ができる．しかし，停止時間を指定できないので，様々な処理を容易に記述するのには適していない．

Stackless Python には，並行処理を記述する方法として，コルーチンと，コルーチンを使いやすくしたタスクレットが使用できる．タスクレットは，1 つのキャラクタに複数の並行処理を記述することも可能で，自律したゲームキャラクタの処理を記述するのに適している．

Stackless Python には，タスクレット間の通信に便利なチャネル機能があり，キャラクタ間通信の記述に使用できる．しかしチャネルは 1 対 1 の通信にしか利用できず，1.9 節の多数の子供キャラクタへの通信記述には適していない．複数のキャラクタが同時に関係する衝突処理のような処理の記述にも適さない．

以上から，Stackless Python は，ある程度の記述機能を持つものの，1.9 節の 4 つの要件を，十分に満足しているとは言えない．

2.2.7　Scheme

Scheme は，関数型の汎用プログラミング言語である Lisp を改良したプログラミング言語である．Scheme や Lisp はシンプルな構文の言語で他の手続

```
(define-state-script ("falling-sign")
  (state ("untouched")
    (on (update)
     [when [task-complete? "wz-post-combat"]
       [go "fallen"] ] )
    (on (event "hanging-from")
     [go "breaking"] ) )
  (state ("breaking")
    (on (begin)
     [spawn-particles-at-joint "self"
       "hinge"
       "sign-break-dust"]
     [wait-animate "self" "sign-break"]
     [go "fallen"] ) )
  (state ("fallen")
    (on (begin)
     [animate "self" "sign-broken"] ) ) )
```

図 2.5: Uncharted2 のサンプル

き型言語等に比較して実装が容易であることから，組み込みやリアルタイムアプリケーション用途への研究や開発も行われている [湯浅 03, DF05, Fje04, ソニ 96, SS98].

PlayStation 用アクションゲーム「Uncharted 2: Among Theives」[SCE09] は，Scheme をベースに Uncharted2 のために開発した独自の言語機構とそのための構文を使用して開発されている [Gre09].

Uncharted2 の開発では，Scheme に次の機能を追加している．

- ゲーム中のキャラクタを 1 つのオブジェクトとして記述する機能
- 状態遷移機械を明示的に記述できる機能
- オブジェクト間の通信や状態遷移の条件に利用できるイベント機能
- 並行処理や処理を一定時間停止する機能
- 並行処理間の通信に使えるシグナル機能

Uncharted2 の Scheme には，ゲーム中のキャラクタを 1 つのオブジェクトとして記述する機能がある．**図 2.5** は Uncharted2 における Scheme プログラムの記述例で，特定のイベントに反応する簡単なキャラクタを `falling-sign` として定義している．

また，キャラクタには状態遷移処理を記述することができる．この `falling-sign` キャラクタの記述内部には，`untouched`，`breaking`，`fallen` の三つの状態が `define-state-script` により定義され，各状態固有の処理が記述されている．状態を遷移させるには，`go` 命令を用いる．

イベント機能は，状態遷移の条件として用いられる．例えば，`untouched` 状態では，`on` に続く `update` と `hanging-from` の二つのイベント時の処理が定義されている．`update` はゲームのアニメーションフレーム毎に発生するイベン

```
(state ("shake-hands")
  (on (begin)
    (track ("player"))
      [wait-move-to "player" "waypoint7"]
      [signal "player-at-waypoint"]
      [wait-for-signal "sully-at-waypoint"]
      [wait-animate "player" "shake-sullys-hand"]
    )
    (track ("sullivan"))
      [wait-move-to "sullivan" "waypoint7"]
      [signal "sully-at-waypoint"]
      [wait-for-signal "player-at-waypoint"]
      [wait-animate "sullivan" "shake-drakes-hand"]
    ) ) )
```

図 2.6: Uncharted2 の track や wait を用いたサンプル

トで，untouched 状態の常に実行される処理を記述している．hanging-from イベントは，他のオブジェクトから発生するイベントであり，発生すると falling-sign オブジェクトの状態は breaking に遷移する．

Uncharted2 の Scheme では，track や wait という記述を用いて，並行処理や時間に沿った処理の記述が可能となっている．track は 1 つの自律した処理を記述する機能であり，**図 2.6** では，1 つの状態 shake-hands の中で，track を用いて 2 つの並行する処理 player と sullivan を記述している．記述内にある，wait-move-to は決められたルールで移動を指示し，移動が終わるまで処理を停止する命令である．ある時間処理を停止する命令にはその他に，アニメーションを再生して，再生終了まで処理を待つ wait-animate 等がある．

並行処理間の通信方法としてシグナルという機能も用意されている．**図 2.6** 中の，[signal "player-at-waypoint"] は，player-at-waypoint という名前のシグナルを送出する命令で，[wait-for-signal "player-at-waypoint"] は，このシグナルを受信できるまで処理を停止する．この仕組みにより track で記述された並行処理の同期が可能で，並行処理間の通信方法として利用することができる．

Uncharted2 の Scheme は，Uncharted2 の開発用に Scheme の柔軟な言語機能を用いて拡張されたもので，Scheme 自体はこの節で述べた機能を持たない．実装には Scheme 処理系についての十分な知見や技術が必要である．

Uncharted2 の Scheme は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 define-state-script や go 等を用いて，状態遷移処理の記述ができる．

要件 2 wait-animate のような，時間に沿った処理の記述が可能な機能がある．

要件 3 track を用いて並行処理を記述することができる．

要件 4 イベントやシグナルを用いて，通信の記述が可能だが，限定された場合にしか利用できない．

Uncharted2 の Scheme には，`define-state-script` の記述中に，`state`，`go` を用いて，状態遷移処理の記述機能が可能である．よって，**図 1.4** のように，変数や条件分岐を用いた，見通しの悪さや間違いの原因となりやすい記述は必要ない．

Uncharted2 の Scheme には，`wait-animate` のような，時間に沿った記述に適した命令が用意されている．

Uncharted2 の Scheme では，個々のゲームキャラクタが自律して処理を進める．また，`track` を用いて，１つのキャラクタに複数の自律した処理の記述も可能である．

Uncharted2 の Scheme では，イベントやシグナルを用いることで，並行処理間の通信も可能となっている．しかし，イベントはイベント名以外の情報を持つことができない．シグナルは処理を停止して通信を待ち受ける並行処理の同期方法としては有効だが，処理を進めながら通信を行うゲームキャラクタ間の通信には適していない．さらに，イベントとシグナルは，個々のキャラクタ内に記述しなければならないので，1.9 節の**図 1.10** として挙げた衝突に適用する場合，通信が個々のキャラクタに分散して可読性を悪くする原因となる．よって，通信機能が十分であるとは言えない．

よって，Uncharted2 の Scheme は 1.9 節の 4 つの要件全てを満たしているとは言えない．

2.2.8 その他の言語

これまで述べたもの以外にも，ビデオゲーム開発やそのスクリプティングを目的としたプログラミング言語や処理系は多い．これらは，細かな違いはあるが本節で説明した言語によく似た記述形式や設計実装がされているため，本節で簡単にまとめて紹介する．

Ruby[まつ 99] は，Python や Perl と同じく汎用のプログラミング言語であるが，いくつかのゲーム開発を目的としたソフトウェア用の記述言語としても使われている．例えば，Star Ruby[星一 08] は Ruby 向けの 2D ゲーム用のライブラリで，Ruby を用いて比較的容易に 2D グラフィクスゲームを記述することができる．また，RPG 制作ソフトウェアである RPG ツクールシリーズ [エン 04] には，Ruby が記述言語として採用されている．RPG ツクールシリーズの Ruby は組み込み用に仕様を軽量化し独自に実装されたものである．

xtal[石橋 07]，GameMonkeyScript[RD03]，AngelScript[Jon01] は Lua や Squirrel に似た，アプリケーションの組み込み目的に設計開発されたプログラミング言語と処理系である．いずれも商用のビデオゲームに利用されており [湊 10]，次の特徴を持つ．

- 手続き型のオブジェクト指向的言語である．

- C/C++プログラムへの組み込みを意識した実装を持つ.
- 状態遷移処理や並行処理の記述に適したコルーチンやマイクロスレッド等の機能を持つ.

これらの言語も Lua や Pawn のようにゲームアプリケーションへの組み込みが考慮されており，軽量でプラットフォームも選ばない設計と実装がされている．どの言語も手続き型のオブジェクト指向言語であるが，Pawn や Stackless Python のような，状態遷移やメッセージの送受信を記述する機能は無い．また，これらの言語のスレッドは Lua と同様に，コルーチンや独自に実装した軽量スレッドを提供するものが多い．これは，コルーチンや軽量スレッドが，手軽に並行処理を記述するのに便利な機能であることと，ビデオゲームプログラムが動作する OS ではスレッド機能を提供していない場合があり，その際にも並行処理を記述できる機能を提供することが理由と考えられる．

Mana[Mor09] は，状態遷移の記述を容易にするために設計実装されたプログラミング言語で，Pawn や Stackless Python に似た，自律的に状態遷移処理を進めるアクターと，アクター同士でメッセージの送受信を記述できるリクエストという機能を持つ．また，アクターは自身の処理内で，`wait` という命令を用いて，一定時間処理を停止する機能もある．

3D Game Studio[CD95] は 3D ゲーム開発を目的とした開発環境で，Lite-C という 3D Game Studio 専用に開発された C 言語に似たプログラミング言語を, ゲームキャラクタの挙動等を記述する用途に提供している．

本節で述べた，Ruby，xtal，Mana，Lite-C 等のプログラミング言語も，1.9 節の 4 つの要件を満たしておらず，ゲームシステム記述に十分な機能を提供していない．

2.2.9　ビデオゲーム開発を目的としたプログラミング言語のまとめ

本節では，ビデオゲーム開発を目的に持つプログラミング言語について述べた．**表 2.1** に説明したプログラミング言語についてまとめた．表の右側 4 項目は 1.9 節の機能的要件をどの程度満たしているかを簡単に ○/△/× で表している．1.9 節で述べたように，4 要件は，ゲームデザイナーがゲームシステムを記述することを容易にするために必要な記述機能として仮定した要件なので，これらの要件を満たさなくともゲームシステムの記述に使用することは可能である．しかし，ゲームシステムの実装が進み，規模が大きく複雑になると，要件を満たさない言語では可読性が悪くなり，修正が容易ではなく，トライアンドエラーの難しい記述になる．

ビデオゲームへの組み込みを目的としたプログラミング言語は，汎用的なプログラミング言語と違い，特定のプラットフォームに依存せず軽量である等，ゲームアプリケーションに組み込まれて利用されることを考慮した設計や実装となっているものが多い．

また，汎用プログラミング言語には無い，状態遷移処理や軽量な並行処理，並行処理間の通信等を記述するための機能を持つものが多い．これらの機能はライブラリではなく，プログラミング言語によって直接提供されているので，組み込んで直ぐに利用可能である．これも特徴の 1 つと考えられる．

しかし，これらのプログラミング言語は 1 章で議論した記述性の要件を満たしているとは言えない．

記述性の要件の 1 つである状態遷移処理の記述機能は，Pawn や Stackless Python 等にはあるが，この機能を有しないプログラミング言語も多い．適切な機能の無い言語の場合，1.9 節の**図 1.4** のように，状態遷移のために変数や条件分岐の記述を必要とするので，簡潔ではない記述となり，誤りの原因となりやすい．

時間に沿った処理の記述を容易にする機能を持つ言語も少ない．Stackless Python や Uncharted2 の Scheme 等にはこの機能があるが，その他の言語は，Lua や Squirrel のように，記述プログラムを毎フレーム呼び出す仕組みが前提となっているものが多く，1.9 節の**図 1.6** のような，時間を管理する変数や条件分岐を必要とし，可読性を悪くし，誤りの原因となりやすい．

また，Lua や Pawn は並行処理の記述機能を持たない．このため，並行処理の記述をするには，**図 2.1** のように処理をフレーム単位で分けて記述したり，組み込むアプリケーション側で相当する機能を用意する等の方法が必要となる[2]．

キャラクタ間の通信については，Pawn のイベントや Stackless Python のチャネル等，機能として組み込まれている言語もいくつかある．しかし，これらは 1 対 1 の通信に限定されていたり，並行処理間の通信の記述には適していない等，ゲームシステム上のキャラクタの関連を表現するのに十分に適したものであるとは言えない．また，多くの言語で利用可能なグローバル変数や手続き呼び出しなどの機能だけでもキャラクタ間の通信は記述可能ではあるが，ゲーム中の様々なキャラクタ間の通信を記述するには，1.9 節の通信の例のように，通信の内容に応じたプログラミングをしなければならず，プログラミングに精通していないゲームデザイナーには難しい．

2.3 ビデオゲーム開発を目的とした統合開発環境

2.3.1 ビデオゲーム開発を目的とした統合開発環境の位置づけ

近年，グラフィカルユーザーインターフェースを利用した，統合的なビデオゲーム開発環境が多数利用出来るようになっている．これらのツールは，前章のプログラミング言語のように組み込まれて動作するのではなく，ゲームアプリケーションを直接生成する．

[2]コルーチン機能を持つ言語は，冗長になるが並行処理を記述することは可能である．

表 2.1: 2.2 節のプログラミング言語

名前	言語の種類	状態遷移	時間の流れ	並行処理	通信
Lua	手続き型	×	×	△ コルーチン	×
Pawn	手続き型	◯State	×	×	△Event
Squirrel	オブジェクト指向	×	×	△ コルーチン	×
Javascript	プロトタイプ方式	×	×	△ コルーチン	×
Stackless Python	オブジェクト指向	×	△schedule	△ コルーチン ◯ タスクレット	△ チャネル
Uncharted2 (Scheme)	関数型	◯state/go	◯wait	◯track	△ イベント シグナル
Ruby (ツクール)	オブジェクト指向	×	×	×	×
xtal	オブジェクト指向	×	×	△ 軽量スレッド	×
GameMonkey	オブジェクト指向	◯states	×	×	×
AngelScript	オブジェクト指向	×	×	×	×
Mana	オブジェクト指向	△ アクター	△wait 命令	△ アクター	△ リクエスト
Lite-C	オブジェクト指向	×	×	×	×

◯ 十分満たしている.
△ 機能として提供されているが十分ではない.
× 機能が提供されていない.

グラフィクスやサウンド処理プログラムの開発には，高度なプログラミング技術と相応の時間を必要とするが，統合開発環境では，グラフィクスやサウンド等の処理プログラムはあらかじめライブラリとして用意されている．そのため，これらのプログラム開発をする必要が無く，開発時間やコストを大幅に削減することができ，ゲームデザイナーにも利用し易い．

また，統合開発環境の多くは，ビジュアルインターフェースを提供するので，操作の習得が容易で，アプリケーションをプログラミングすること無く，ある程度制作可能である．ゲーム画面を直接視認しながら制作を進められる機能を持つものもある．

しかし，ゲームシステムの全ての要素をビジュアルインターフェースで記述することは難しい．そこで詳細な記述にも対応できるように，ビジュアルインターフェースによる記述機能の他に，プログラミング言語を用いた記述を可能にしているものもある．

本節では，ビデオゲーム開発を目的とした統合開発環境のうち，商用に利用されているか広く知られているものについて，その機能や特徴を説明し，1.9 節の 4 つの要件についての評価を行う．また，最後の 2.3.8 節では，説明した統合開発環境の評価をまとめる．

2.3.2　Unity

Unity[Tec06] は，3D のリアルタイムゲーム開発を目的とした統合的な開発環境である．比較的安価でありながら高度な機能を持つことと，近年のスマートフォンへの対応等により，広く利用されるようになってきている．記述言語として JavaScript を利用できるが，他に C#[ECM06] や Python[Lut98] 等の言語も利用可能である．

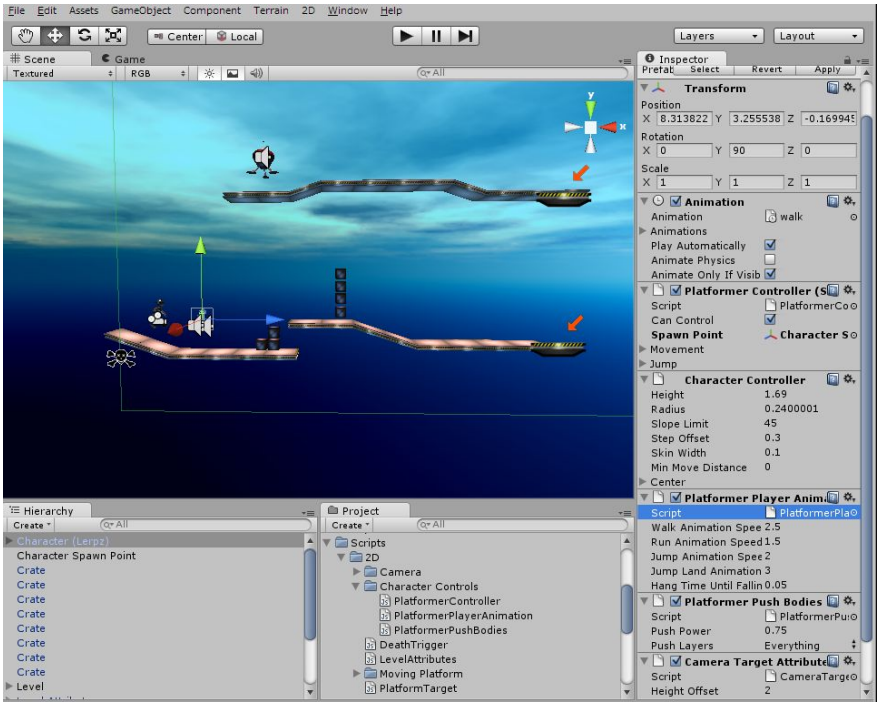

図 2.7: Unity の画面

Unity では，ゲーム中のキャラクタや背景の配置はグラフィカルなユーザーインターフェース上で行う (**図 2.7**)．あらかじめ簡単な JavaScript や C#のサンプルが準備されているので，衝突時に音を出したり，移動方向を変えるなどの簡単な処理の追加や修正であれば，記述言語についての知識なく進めることが可能である．

図 2.8 は Unity で JavaScript を用いたときの記述のサンプルである．Unity では，配置したキャラクタやオブジェクトごとに JavaScript を用いた挙動の記述が可能である．毎フレームの処理や，衝突時の処理は，決められた名前の関数を定義することで記述可能である．例えば，**図 2.8** には `Start`，`Update`，`OnControllerColliderHit` の 3 つの関数が定義されており，それぞれキャラクタが登場した時の処理，毎フレームの処理，何かに衝突したときの処理に対応している．

これらの関数は，キャラクタやオブジェクトの種類等にも依存するが，様々なものが用意されている．また，アニメーションやサウンド等多くの機能が命令として用意されており，ユーザーは，Unity 上で高度な 3D アプリケーションを開発することが可能となっている．

このように，Unity は高度なグラフィクス，衝突判定処理やサウンド等の機能があらかじめ用意されており，グラフィカルなユーザーインターフェースのみでも，簡単なインタラクティブアプリケーションを容易に制作することができる．3D グラフィクス用のツールで制作したデータを使用するため

```
// キャラクタのデータや設定
var runAnimationSpeed = 1.5;
    :
// キャラクタ登場時の初期化の処理
function Start () {
  animation.Stop();
    :
}
// アニメーションフレームごとの処理
function Update () {
  var controller:PlatformerController = ...
  // 立っているときの処理
  if (controller.GetVelocity() < runAnimationSpeed) {
    :
  // 走っているときの処理
  } else {
    :
  }
}
// 衝突時の処理
function OnControllerColliderHit(hit:ControllerColliderHit) {
    :
}
```

図 2.8: Unity の JavaScript による記述のサンプル

のツールも揃っているので，それらを処理するための高度なプログラムを開発することなく，3D グラフィクス等を駆使したゲームアプリケーションの開発も可能で，開発時間を大幅に短縮することができる．他の同程度の機能性を持つツールと比較して安価で，無料版もあり，ツールについての議論や情報交換が自由であることから，ドキュメントや情報も豊富で，近年では多くの商用のゲームアプリケーションが Unity を用いて開発されるようになっている．

Unity は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 キャラクタの `Update` が毎フレーム呼び出される規約によりキャラクタ単位の並行処理を記述可能で，コルーチンを使用することも可能だが，十分ではない．

要件 4 あらかじめ用意されている衝突等のイベント時に呼び出される手続きの規約はあるが，限定的で，キャラクタ間の様々な通信に利用することが難しい．

Unity で使用できるプログラミング言語は，JavaScript や C#等の汎用的なプログラミング言語である．これらの言語には，状態遷移処理を記述する

ための機能は無く，状態を表す変数や条件分岐を用いて記述しなければならない．

また，時間に沿った処理の記述に適した機能も無いので，**図 1.6** のように，時間を表す変数や条件分岐を用いて記述しなければならず，見通しが悪く間違いの原因となりやすい．

Unity では，各キャラクタの `Update` メソッドが毎フレーム呼ばれる規約となっており，キャラクタ単位での並行した処理の記述は難しくない．しかし，フレーム単位でしか呼び出されないことと，1 つのキャラクタに複数の並行する処理を記述することはできないので，要件を十分に満足してはいない．また，Unity では JavaScript や C#のコルーチンを使った記述が可能だが，2.6.3 節で議論するように，コルーチンは自律したゲームキャラクタの処理を記述することには適していない．

Unity には，衝突やユーザーの入力等をイベント時に呼び出される `OnControllerColliderHit` 手続き等の規約があり，キャラクタ間の衝突等の通信をある程度記述しやすい仕組みとなっている．しかし，Unity が提供している規約は，衝突やユーザーの入力等の特定のイベントに限定されており，キャラクタ間の様々な通信を記述するには不十分である．

以上から，Unity は 1.9 節の 4 つの要件を十分には満たしていない．

2.3.3 Flash と Action Script

Flash[Ado] は Adobe 社が提供する，Web ブラウザ上でベクターグラフィクスやアニメーション等のアプリケーションを提供可能にするアプリケーションプラットフォームである．

Action Script[SR08] は，Flash アプリケーションを記述するために提供されるプログラミング言語で，JavaScript の標準である ECMAScript[ECM99] を拡張したプロトタイプ方式のプログラミング言語である．

単純な動画再生や，比較的簡単なユーザー入力を処理する Flash アプリケーションであれば，Action Script を用いて記述することなしに開発することも可能であるが，Action Script を使用すれば，よりインタラクティブ性の高いアプリケーションの開発が可能である．

ブラウザ上で手軽に動作させることが可能なことと，高度なグラフィクスやユーザー入力等の処理を使ったアプリケーションを容易に開発できることから，Action Script を用いて記述されたゲームアプリケーションは多い．モバイル機器用に Flash Lite という規格もあり，多くのモバイル機器で採用されていることからモバイル機器上でのゲームアプリケーション開発にも利用されている．また，開発ツールが充実しており，学習や開発上の問題解決のための書籍や情報も多い．

Action Script を使用可能な Flash アプリケーションの開発環境としては，無償の Flash Develop[MP05] や MTASC[MT01] 等も知られているが，ここ

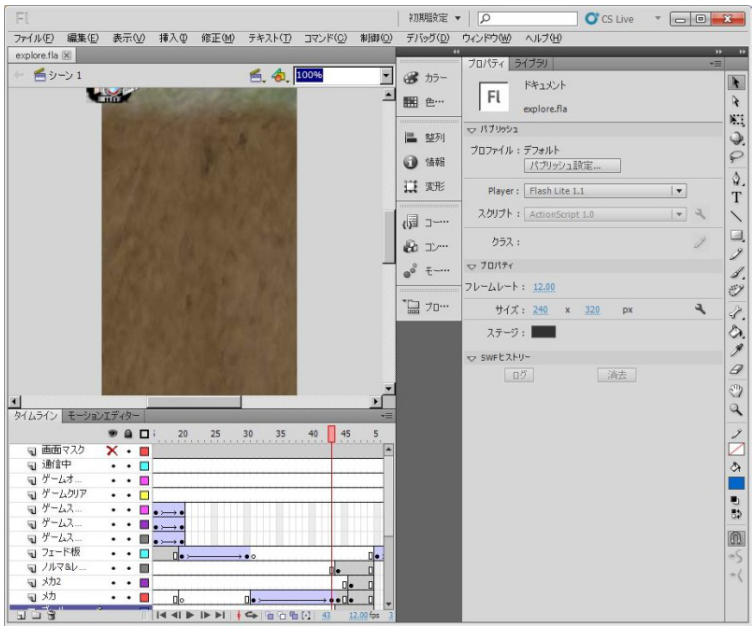

図 2.9: Adobe Flash CS5 の画面

では，統合開発環境として良く整備されていることから，Adobe 社が提供している Adobe Flash CS5 を取り上げる．

Adobe Flash 上での開発は，Unity と同じく，グラフィカルインターフェースを持ち (**図 2.9**)，特に Action Script を使った記述の必要なしに，キャラクタやオブジェクトの生成や配置を行うことができる．

Flash は Web ブラウザにアニメーション機能を導入するためのものであったことから，アニメーションの時間の流れに沿った処理の記述も容易である．**図 2.9** の下方にあるのがタイムライン編集部分であるが，表示オブジェクトごとに横方向にタイムラインを持ち，このタイムライン上で直感的に動作や表示の編集が可能である．タイムライン上の任意の場所に，Action Script を使って記述したプログラムを埋め込むことで，Action Script プログラムを任意のタイミングで実行させることもできる．実行時，すべてのタイムラインは同時に並行して処理を開始する．

Action Script と Adobe Flash CS5 は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 タイムラインを用いて時間の流れにそった記述が容易である．

要件 3 オブジェクトは固有のタイムラインを持ち，並行処理の記述も容易である．ただし，1 つのオブジェクトが複数のタイムラインを持つことはできない．

要件 4 キャラクタ間の通信に適した機能は無い.

Action Script には，状態遷移処理の記述に適した機能は無く，状態を表す変数や条件分岐を用いて，**図 1.4** のように記述しなければならず，見通しの悪さや間違いの原因となりやすい.

Adobe Flash CS5 には，タイムラインを用いて時間に沿った処理を記述する機能がある．アニメーションだけでなく，ゲームのキャラクタの挙動に使用することもできる.

Adobe Flash CS5 上では，各オブジェクトが固有のタイムラインを持ち，各タイムラインは起動と同時に自律して処理を進める．このため，オブジェクト毎の並行処理の記述が容易である．しかし，1 つのオブジェクトが複数のタイムラインを持つことはできないので，1 つのオブジェクトに複数の並行処理を記述するには，**図 1.8** のように，1 フレームづつ処理を進める手続きを複数記述して，毎フレーム呼び出すような記述や，複数のオブジェクトを組み合わせて 1 つのオブジェクトを表現するなどの工夫が必要となる.

Action Script には，オブジェクト同士の通信の記述に適した機能は用意されていない．このため，通信を記述するには，**図 1.9** や**図 1.10** のように，通信は処理内容に応じて様々な記述方法を使い分けなければならず，キャラクタの通信をわかりやすく明示することが難しい.

以上から，Adobe Flash CS5/Action Script は 1.9 節の 4 つの要件を十分には満たしていない.

2.3.4 CryEngine

CryEngine[CRY04] は，商用の 3D のリアルタイムゲーム開発，特に First Person Shooting(FPS) と呼ばれる一人称視点のシューティングゲーム開発を目的とした統合開発環境である．FPS ゲームでは，**図 2.10** のようにゲーム画面の視点がゲームフィールド中のプレーヤー自身であり，画面にプレーヤーは表示されない．プレーヤーは 3D ゲームフィールド内を移動して，現れる敵キャラクタの攻撃をかわしながら銃等の武器を用いて倒すことを目的とする.

CryEngine は多くの商用ゲーム開発で利用されており，高機能かつ高性能なことで知られている．CryEngine は，元は Crytek 社が 2004 年に開発した Far Cry[UBI04] (**図 2.10**) という FPS ゲームのフレームワークであったが，その有用性から開発環境として単体で販売されるようになった.

CryEngine には記述言語として，前節で述べた Lua が採用されている．CryEngine の Lua には，CryEngine で実装されている高度なグラフィクスや物理計算エンジン等の様々な機能を利用するための API が多数用意されており，利用者は Lua を使って様々な 3D ゲームの開発が可能である．また，ゲーム中のキャラクタや背景などの配置や形状，ある程度の衝突時の処理などは Lua による記述なしに，グラフィカルインターフェースのみを用いて開発することができる.

図 2.10: Far Cry のゲーム画面

CryEngine も Unity と同様に，各キャラクタの決められた Lua の手続きが毎フレーム呼ばれる規約により，キャラクタ単位での並行処理記述が可能となっている．また，特定のゲーム中のイベント発生時に呼ばれる手続きの規約等もある．

最新の CryEngine では Lua を用いたテキスト形式での記述の他に，フローグラフと呼ばれるビジュアルプログラミング環境も提供している (**図 2.11**)．これは Lua プログラムのラッパーのように機能し，Lua で直接記述しなくとも，変数や制御構造をグラフィカルに編集することが可能である．

CryEngine は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 キャラクタごとの並行処理の記述が可能であるが不十分．

要件 4 あらかじめ用意されている衝突等のイベント時に呼び出される手続きの規約はあるが，限定的で，キャラクタ間の様々な通信に利用することが難しい．

CryEngine は，記述に使用するプログラミング言語が Lua であり，Lua には，1.9 節の 4 つの要件を十分に満たす機能が用意されていない．状態遷移処理を記述する機能が Lua には無いので，状態を表す変数や条件分岐を用いて記述しなければならない．

時間に沿った処理の記述方法も用意されていないので，**図 1.6** のように，時間を表す変数や条件分岐を用いて記述しなければならず，見通しが悪く間違いの原因となりやすい．

CryEngine が Lua に追加する，毎フレーム呼び出される手続きやイベント処理時に呼び出される手続き等の規約により，キャラクタ単位の並行処理や，

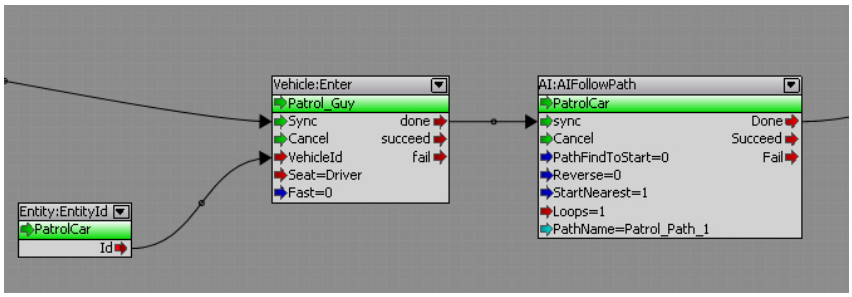

図 2.11: CryEngine のフローグラフ

特定のゲームイベントを処理の記述が可能であるが，1 つのキャラクタに複数の並行した処理を記述できず，イベントは衝突や入力のように CryEngine が用意した機能に限定されており，十分とは言えない．

2.3.5 Tonyu

Tonyu[長慎 01] は 2D グラフィクスのアクションゲーム開発を目的とした開発環境である．Tonyu もゲーム中のキャラクタの配置をグラフィカルインターフェース上で行うが (**図 2.12**)，その挙動は，独自の記述言語を用いて記述する．

記述言語は，動的型付けのオブジェクト指向言語で，キャラクタごとに 1 つのクラスとして記述する．クラスは 1 つのスレッドを持ち，クラス単位の並行処理記述を行う．メソッドの定義も可能で，特定の名前のメソッドはゲーム中のイベント等の処理記述用に再定義することができる (**図 2.13**)．

キャラクタの処理記述内では，`update()` を使用することができる．これは，2.2.6 節で述べた Stackless Python が提供する `stackless.schedule()` と同じ機能で，キャラクタの処理が `update()` まで来ると，そのフレームの処理は終了し，次のフレームは，停止した `update()` の次から処理を再開する．

Tonyu は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 `update()` で時間に沿った処理をある程度容易に記述できる．

要件 3 キャラクタ単位で並行処理の記述が可能だが，1 つのキャラクタに複数の並行処理を記述することはできない．

要件 4 キャラクタ間の通信を記述するのに適した機能がない．

Tonyu には，状態遷移処理を記述するため機能は無く，状態を表す変数や条件分岐を用いて記述しなければならない．

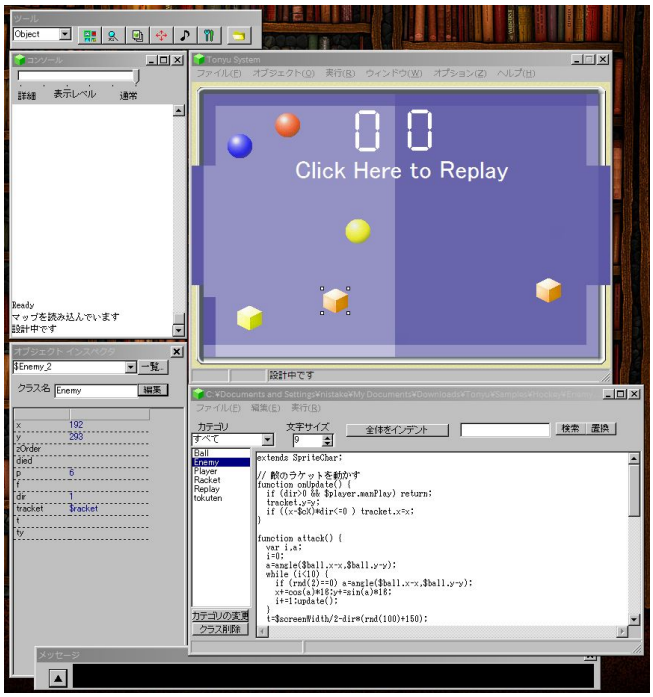

図 2.12: Tonyu の画面

Tonyu では，update() を用いて，1 フレーム時間分だけ処理を停止する記述ができる．しかし，停止時間を指定できないので，様々な処理を容易に記述するのには適していない．

Tonyu は 1 つのキャラクタが 1 つの自律した処理を持つ．また，Tonyu では，Unity に似た，各キャラクタの onUpdate や onDraw 手続きが毎フレーム呼ばれる規約となっていて，これも並行処理の記述に利用できる．しかし，1 つのキャラクタに複数の並行する処理を記述することはできない．

Tonyu には，キャラクタ間の通信の記述に適した機能はない．キャラクタ間の様々な通信を記述するには，1.9 節で述べたように，通信の方法や内容によって異なる記述をしなければならず，わかりやすく明示することが難しい．

以上から，Tonyu は 1.9 節の 4 つの要件を十分に満たしていない．

2.3.6　吉里吉里

プレーヤーがコマンド入力しながら用意されているシナリオを進めていくゲームは，シナリオゲームやノベルアドベンチャーゲームと呼ばれる．シナリオゲームは，分岐や合流を繰り返しながら進むシナリオがゲームの中心的なロジックであることから，シナリオ記述専用の言語を用いて開発される [清木 04]．この記述言語とグラフィカルなユーザーインターフェースを組み合わせた，シナリオゲーム開発環境は少なくない．

```
extends SpriteChar;  // 継承

// このキャラクタの追加の情報
n = 0;
life = 100;

// キャラクタのスレッドの処理
while( 1 ) {
    :
    // このキャラクタの処理を行う
    :
  update(); // オブジェクトの書き換え処理
}

// update() が呼ばれたときの処理
function onUpdate() {
}
// 描画時の特別な処理
function onDraw() {
  // 配置したオブジェクト以外で表示に必要な処理を記述
}
// マウスボタンが押されたときの処理
function onMouseDown( c ) {
}
```

図 2.13: Tonyu の記述サンプル

吉里吉里 [W.De03] は，そのようなシナリオゲーム開発環境と記述言語である．吉里吉里は，TJS という独自のオブジェクト指向言語と KAG というシナリオ進行を記述する言語の二つを記述に用いる．TJS は汎用的なオブジェクト指向言語に似た文法の言語で，画面表示，入力，音などの機能の記述や実装に適している．KAG はシナリオの流れを分岐も含めて直感的に記述することができる．吉里吉里は二つを組み合わせることでシナリオゲームの記述性を高めている．

また，吉里吉里には，ゲーム中に使用するグラフィクスやサウンドなどのリソースを管理する機能や，簡単なデバッガ機能なども用意されている．

図 2.14 は吉里吉里のサンプルで，りんごを 10 個食べるとゲームオーバーになる単純なシナリオを記述している．プログラム内で [] の内部が，TJS スクリプトの実行を表し，それ以外が KAG によるシナリオ記述となっている．

*start，*eat 等の*で始まる行は KAG のラベル定義で，シナリオの開始位置や分岐点の指定をする．シナリオの分岐処理や変数の設定は [] 内の TJS スクリプトで記述している．

プレーヤーの選択による分岐は*start で，link を使用して記述されている．プレーヤーの選択は「食べる」と「捨てる」の二択となっている．りんごを食べた個数による分岐は，*eat で if と変数 f.num を用いて記述されており，10 回「食べる」とゲームオーバーとなる．

```
[wait time=200]
[eval exp="f.num = 0"]

*start|りんごがあるよ
[cm]
[link target=*eat] 食べる [endlink][r]
[link target=*dispose] 捨てる [endlink][r]
[s]

*eat
りんごを食べるよ．むしゃむしゃ．[l]
[eval exp="f.num = f.num + 1"]
[if exp="f.num < 10"]
むしゃむしゃ．おいしい．[l]
[jump target=*start]
[else]
しまった！食べ過ぎた！[l]
[jump target=*gameover]
[endif]

*dispose|りんごを捨てた
捨てるとまたりんごが降ってきた！[l]
[jump target=*start]

*gameover|ゲームオーバー
食べすぎにはご用心.
[s]
```

図 2.14: 吉里吉里のサンプル

吉里吉里にはこのほかにも，記述を簡潔にしたり，同一構造のシナリオコードを再利用可能にするためにサブルーチンやマクロ機能が用意されている．

吉里吉里は，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能が無い．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 並行処理の記述に適した機能が無い．

要件 4 キャラクタ間の通信の記述に適した機能が無い．

吉里吉里は，ユーザーの入力に応じてストーリーが展開していくというシナリオゲームに特化した開発環境であるので，自律して状態遷移をしながら動作するゲームキャラクタを記述する機能が無い．そのため，1.9 節のいずれの要件も満たしていない．

2.3.7　その他の統合開発環境

統合的なゲーム開発環境としては，これまでに述べたもの以外に，3D GameStudio [CD95]，XNA Game Studio[Cor]，HSP[悠黒 01] や GameClosure [CHFL11]

等が挙げられる．

3D Game Studio[CD95] は Unity や CryEngine と同じく，3D ゲーム開発を目的とした開発環境で，ゲームキャラクタのロジックを記述する言語として，Lite-C や C-Script という，C 言語に似た言語を独自に設計開発して利用している．

XNA Game Studio[Cor] は Microsoft 社によるゲーム開発のための開発環境で，同社の C++や C#言語用の開発環境である Visual Studio[MCb] と，高度なグラフィクスや物理演算ライブラリなどを含むリアルタイムゲーム開発に特化した XNA Framework を統合したソフトウェア群である．XNA Game Studio 上での開発には，C#[ECM06] が利用されることが多い．

HSP[悠黒 01] は，Windows 上のインタラクティブなアプリケーションを記述できるプログラミング環境である．規模の小さなアプリケーションであれば容易に開発可能なので，HSP を利用して制作されたゲームも多い．記述には HSP 独自の BASIC によく似た比較的平易なプログラミング言語を使用する．

GameClosure[CHFL11] は，HTML5 を利用したブラウザ上で動作するゲーム開発を目的としたゲーム開発環境である．グラフィカルなユーザーインターフェースを用いて，ゲーム中のオブジェクトの配置等を行い，JavaScript を用いて動きやゲームルールの記述を行う．

シナリオゲーム用の開発環境には，SCUMM[Luc87]，NScripter[高橋 09]，Black Diamond[小野 01]，System3/System4[SOF] 等，多数のソフトウェアが知られている．

シナリオゲーム用の開発環境での記述は，if-then-goto 形式でシナリオを分岐処理させていく形式となっているが，これは，シナリオの進行と分岐を表現しやすく実装が容易な反面，複雑なシナリオを記述すると管理の難しいプログラムとなりやすい [清木 04]．また，シナリオゲームの記述ツールは，複数のキャラクタが登場し，様々な通信をするゲームの記述には適していない．

2.3.8 ビデオゲーム開発を目的とした統合開発環境のまとめ

本節では，組み込みを必要としない，ビデオゲーム統合開発環境について述べた．本節で述べた開発環境を**表 2.2** にまとめた．表の右側 4 項目は 1 章で述べた機能的要件をどの程度満たしているかを簡単に ○/△/× で表している．

1.9 節で述べたように，4 要件は，ゲームデザイナーがゲームシステムを記述することを容易にするため必要な機能として仮定した要件であり，要件を満たさなくともゲームシステムの記述は可能である．しかし，ゲームシステムの実装が進み，規模が大きく複雑になると，要件を満たさない言語では可読性が悪くなり，修正が容易ではなく，トライアンドエラーの難しい記述になる．

ビデオゲーム統合開発環境は，多くがグラフィカルなインターフェースを持ち，一般的なプログラミング言語を用いて記述するより容易にビデオゲー

表 2.2: 2.3 章の統合開発環境

名前	プラットフォーム	目的ゲーム	記述言語	状態遷移	時間	並行処理	通信
Unity	Windows その他	リアルタイム	JavaScript C#	×	×	△	×
Flash	Web ブラウザ モバイル	-	Action Script	×	○	△	×
CryEngine	Windows	主に FPS	Lua	×	×	△	×
Tonyu	Windows	リアルタイム	独自言語	×	△	△	×
3D GameStudio	Windows	リアルタイム	Lite-C	×	×	△	×
XNA Game Studio	Windows	-	C#	×	△	△	×
HSP	Windows	-	独自言語	×	×	×	×
GameClosure	Web ブラウザ	-	JavaScript	×	×	×	×
吉里吉里	Windows	シナリオゲーム	独自言語	×	×	×	×
SCUMM	Windows その他	シナリオゲーム	Lua	×	×	×	×
NScripter	Windows	シナリオゲーム	独自言語	×	×	×	×
Black Diamond	Windows	シナリオゲーム	独自言語	×	×	×	×

○ 十分満たしている.
△ 機能として提供されているが十分ではない.
× 機能が提供されていない.

ム開発が可能である.

統合的なインターフェースを持っていても，詳細な記述はプログラミング言語でなければ難しい場合があることから，どのツールも記述用のプログラミング言語を持っている．これらのツールは，前章のプログラミング言語が組み込まれた，より高レベルの開発ツールであるともみなせる.

ビデオゲーム統合開発環境は，グラフィクスやサウンド等をツールが提供しなければならないため，対応可能なプラットフォームは限定されたものが多い．多数のプラットフォームに対応した Unity は，この点で他のツールよりも優れている.

グラフィクスやサウンド機能の実装には高度なプログラミング技術が要求され，ゲームデザイナーが行うのは難しい．本節で説明したビデオゲーム統合開発環境は，これらの機能を提供しているので，ゲームデザイナーが使用してゲームシステムの開発をすることが可能である.

リアルタイムゲームを目的としたツールは，オブジェクト指向型のプログラミング言語を記述言語としているものが多い．これは，オブジェクト指向言語を使用すると，ゲーム中のキャラクタの処理の記述や，グラフィクスやサウンド等の機能の提供がしやすくなることが理由であると考えられる.

シナリオゲームの場合は，シナリオの記述の容易さから，IF-THEN 形式の独自言語を提供しているものが多く，高度なプログラミングの経験がなくともシナリオの記述が可能である．しかし，自律して動作するキャラクタが通信をしながらゲームを進めるというモデルを前提としないので，1.9 節で議

論した 4 つの要件を満たさない.

本節で述べた開発環境は，ゲーム開発に有用な機能を多く採用することで開発を容易にしているが，1 章で述べた要件を必ずしも満たしていない.

例えば，Unity や CryEngine は，記述言語として Lua や JavaScript を用いており，これらのプログラミング言語と同様の記述機能不足の問題を持つ.

Adobe Flash はタイムラインを用いて 1 章で述べた要件 2 の時間に沿った処理や並行処理は比較的平易に記述可能だが，状態や並行処理間の通信に適した記述機能を持たない．HSP や Tonyu も状態の記述や通信に適した機能を持たない.

また，キャラクタ間の通信の記述に適した機能を提供しているツールは無く，ゲームキャラクタの相互関係を記述するためには，Lua や JavaScript などの言語機能だけを用いなければならない.

2.4 個々のゲーム開発専用に作られた記述ツール

2.4.1 個々のゲーム開発専用に作られた記述ツールの位置づけ

ビデオゲーム開発では，ゲームシステム記述ツールを独自に設計開発することも少なくない [西森 03]．記述ツールを特定のゲーム専用に特化した設計や実装にすることで、目的とするゲーム記述に関して優れた記述機能を提供したり、開発するゲームプログラムに適した実装にすることができる。また，開発プロジェクト専用に用意するので，記述ツールを利用するプロジェクト内のゲームデザイナーにとって使いやすい設計とすることもできる．本節では，これら個々のゲーム専用に開発された記述ツールについて説明する.

これらの専用の記述ツールは，ユーザーに後から利用されることを目的とするために，その実装や詳細が公開されているものと，ある製品の開発内部で使用されることを目的とし，詳細が公開されていないものがある.

本節で説明する QuakeC や Unreal Script は記述ツールが公開されているもので，ユーザーに製品購入後にゲームの修正を可能にさせること目的としている．先に述べた Pawn を利用した Grand Theft Auto も，記述ツールが公開されている製品の 1 つであるが，Pawn 自体は Grand Theft Auto のために開発されたプログラミング言語ではないので，この分類には含めていない.

一方，多くのビデオゲーム製品は，ソースプログラムや開発手法が公開されることは少なく，記述ツールについての情報はほとんどない．公に発表されることは多くないため，特定のゲーム専用に開発された記述ツールや言語の詳細が公開されることは稀である．そこで，ビデオゲーム開発を行っている企業に，資料の提示を依頼する形式で，独自に記述ツールに関する調査を行った．制作物の多くが社外秘となる企業が多いことから，調査資料として利用できるのは，5 社 6 タイトルの記述ツールであった.

表 2.3 に本節で説明する記述ツールをまとめた．非公開の記述ツールについては，K1 や S1 のようにそのタイトルや詳細を伏せて表記した.

表 2.3: 個々のゲーム専用の記述ツール

記述言語名	ゲーム名	ゲームのタイプ	プラットフォーム
QuakeC	Quake	FPS	Windows
Unreal Script	Unreal Tournament		
非公開	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

∗ 非公開のため区別目的で本論文でつけた仮の名前

本節では，まず，実装や詳細が公開されている QuakeC と UnrealScript について，機能や特徴について説明し，1.9 節の 4 つの要件についての評価を行う．その後，詳細が公開されていない**表 2.3** の「非公開」の記述ツールに関して，機能や特徴の説明と，1.9 節の 4 要件の評価を行う．最後に，2.4.7 節で，これらの記述ツールの評価についてまとめる．

2.4.2　QuakeC

Quake[iS96] は，2.3.4 節で述べた CryEngine を利用した Far Cry と同様の FPS と呼ばれる一人称視点のゲームである (**図 2.15**).

図 2.15: Quake のゲーム画面

Quake はソースプログラムが公開されており，その中には，キャラクタの振る舞いやパラメータを変更するスクリプト言語システムが内包されている．

ユーザーは，これを利用して既存のキャラクタや武器の振る舞いを変えたり，独自の新しいキャラクタを追加することが可能である．

記述には QuakeC[Mon96] と呼ばれる C 言語に類似した Quake 専用に開発されたプログラミング言語を使用する．QuakeC は Quake ゲーム専用の記述システムであるため，Quake 無しに動作させることはできない．

QuakeC には次の特徴がある．

- 手続き指向なプログラミング言語である．
- 基本的な数値データ型，変数，制御構造を記述する機能を持つ．
- キャラクタは 1 つのスレッドを持ち，自律的に処理を進める．
- キャラクタが処理する関数を切り替える機能により，状態遷移処理の記述が可能である．
- 衝突などの定義済みのイベントに対応した関数を定義することで，衝突発生時の処理記述が可能になっている．

QuakeC は，C 言語の関数にあたる手続きを列挙してプログラムを記述する，手続き指向のプログラミング言語である．**図 2.16** は QuakeC のゲーム中のキャラクタの記述例であり，`player_stand` や `player_run` で始まる名前の手続き群でプログラムが構成されている．

QuakeC は，データ型として整数や実数を利用することができる．また，指定したデータ型の変数や `if` や `while` 等の制御構造を使用することもでき，比較的自由度の高いカスタマイズが可能である．

QuakeC では，1 つのキャラクタが 1 つのスレッドを持つ．スレッドは毎フレーム関数を呼び出す仕組みになっており，各キャラクタが自律して処理を進める並行処理の記述が可能になっている．

図 2.16 には「立ち」と「走り」の 2 つの状態を持つキャラクタが記述されている．2 つの状態は，`player_stand` で始まる名前の手続き群と，`player_run` で始まる名前の手続き群で構成され，立ち状態であれば，スレッドはフレーム毎に，`player_stand1`，`player_stand2`，`player_stand3` と呼び出す関数を切り替えながら処理を進める．次のように [] 内に次に処理する関数を指定することで，処理する関数の切り替えを記述できる．

```
void()  player_stand1 = [ $stand1, stand2 ]
```

各手続き内部では，`self` という予約後を用いて，処理しているキャラクタの位置や速度等の属性の参照や設定ができ，個々のキャラクタの動作の記述が可能である．立ちや走りのアニメーションのコマに対して 1 つづつ処理を記述するため，状態毎にアニメーションコマ数分の関数が必要となり，記述量が多く冗長なプログラムとなりやすい．実際のゲームでは，同一状態であればどのコマでも同一の処理をすることが多いので，立ち状態や走り状態を処理する関数を別途記述して，それを呼び出すことで記述量を減らす．

```
/* プレーヤー立ち状態 (アニメーション 1 コマ目)
 * [] 内は，一つ目が再生するアニメーションコマ名,
 * 二つ目が，次のアニメーションコマ名.
 */
void()  player_stand1 =[  $stand1,  stand2 ]
{
  /* 移動速度が設定されているなら走り状態にする */
  if (self.velocity_x || self.velocity_y) {
    self.think = run1;  /* 次のコマを強制的に run1 に変更 */
  }
};
/* プレーヤー走り状態 (アニメーション 2 コマ目以降) */
void()  player_stand2 = [  $stand2,  stand3 ] { ... }
void()  player_stand3 = [  $stand3,  stand4 ] { ... }
void()  player_stand4 = [  $stand4,  stand5 ] { ... }
void()  player_stand5 = [  $stand5,  stand6 ] { ... }
  :
  :
/* プレーヤー走り状態 (アニメーション 1 コマ目) */
void()  player_run1 =[  $run1,  run2 ]
{
  /* 移動速度が 0 なら立ち状態にする */
  if (!self.velocity_x && !self.velocity_y) {
    self.think = stand1; /* 次のコマを強制的に stand1 に変更 */
  }
};
/* プレーヤー走り状態 (アニメーション 2 コマ目以降) */
void()  player_run2 = [  $run2,  run3 ] { ... }
void()  player_run3 = [  $run3,  run4 ] { ... }
void()  player_run4 = [  $run4,  run5 ] { ... }
void()  player_run5 = [  $run5,  run6 ] { ... }
  :
  :
```

図 2.16: QuakeC のサンプル

衝突等の特定のゲーム中のイベントは，それに対応した関数を定義する規約となっており，プログラム中でイベントに応じた処理の記述が可能になっている．

QuakeC は 1.9 節の 4 つの要件について，次のように評価できる．

要件 1 手続きをキャラクタの状態として記述することが可能だが冗長.

要件 2 時間に沿った処理の記述に適した機能が無い.

要件 3 並行処理はキャラクタ単位でのみ可能.

要件 4 キャラクタ間の通信の記述に適した機能が無い.

QuakeC は，関数と処理する関数の切り替えにより，状態遷移処理の記述が可能となっている．しかし，各状態のフレーム毎に関数を記述しなければならず，冗長な記述になりやすく，見通しの悪さや誤りの原因となるため，十分に要件を満たしているとは言えない.

QuakeC では，時間に沿った処理の記述が難しい．あるフレームで実行したい処理は，そのフレームに対応する関数にその処理を記述することはできるが，わかりやすい記述方法ではない．

各キャラクタは 1 つのスレッドを持つので，キャラクタ単位の並行処理の記述は可能である．しかし，1 つのキャラクタに複数の並行処理を記述することはできない．

QuakeC には，キャラクタ間の通信に適した記述機能が用意されていない．キャラクタの通信を実装するには，変数や制御構造等を使って，処理に応じた異なる記述が必要となり，キャラクタの通信をわかりやすく明示することは難しい．

以上から，QuakeC は 1.9 節の 4 つの要件を十分には満足していない．

2.4.3 UnrealScript

UnrealScript[EG02] は，Unreal Tournament という Quake と同様の FPS ゲーム用に開発されたスクリプト言語である．Unreal Tournament 製品パッケージに，ゲーム改変用ツールとして同梱されている．

UnrealScript は QuakeC よりも言語の機能が豊富であることから，QuakeC よりもさらに高度な修正が可能である．UnrealScript を利用して，元の Unreal Tournament とは異なるルールのゲームを作ることも可能である．

QuakeC 同様，UnrealScript は Unreal Tournament プログラム上でのみ利用可能な記述システムであるため，Unreal Tournament プログラムではない，別のアプリケーションプログラム上で利用することはできない．

図 2.17 は UnrealScript の例である．UnrealScript の特徴を次に挙げる．

- オブジェクト指向プログラミング言語である．
- キャラクタを定義するための Actor という定義済みクラスがある．1 つの Actor は 1 つのスレッドを持つ．
- キャラクタの状態遷移を記述する State という機能があり，State ごとにゲーム中の衝突などに対するイベント処理を記述することができる．
- State の定義内では `Sleep` 等の命令を使い，時間に沿った処理の記述ができる．

UnrealScript はオブジェクト指向言語であり，変数や関数を持つクラスをプログラムの記述単位とする．ライブラリはクラスで整備されており，利用者は記述したいキャラクタや処理に適した UnrealScript 組み込みのクラスを継承して記述を行う．

UnrealScript はキャラクタを記述するのに Actor クラスという組み込みクラスを利用する．Actor クラスは全てのキャラクタの基本となるクラスで，衝突やキー入力などのイベント処理を記述しやすくする機能を持つ．

```
class Player extends Actor;
auto state start {
Begin:
  GotoState('Idle');
}
state Idle {
  function Touch( actor Other ) { // 他のキャラクタに触れた時の処理
    GotoState('Attacking');
  }
Begin:
  sleep(10);
  goto 'Begin'; // C/C++の goto と同じ．GotoState ではない.
}
state Attacking {
Begin:
  if ( ... ) {  // 何らかの攻撃状態終了条件
    GotoState('Idle');
  }
  sleep(10);
  goto 'Begin';
}
```

図 2.17: UnrealScript のサンプル

Actor クラスには，状態遷移を記述するための State という機能がある[3]．**図 2.17** の state() として定義される関数は，キャラクタの 1 つの状態を表す．状態遷移は，State を変更する GotoState という命令を使用する．個々の State では固有のイベント処理を記述することも可能である．例えば，**図 2.17** の Idle には，Touch という他のキャラクタに触れたときに呼び出される関数が定義されていて，この関数の実行後は，Attacking へ状態が遷移する．

State の定義内では，Sleep 等の命令により一定時間処理を停止させることができる．これにより時間の流れに沿った処理の記述が可能になっている．複数の並行する処理が 1 つの関数を同時に実行すると発見の難しい不具合の原因となる場合があるので、Sleep 等の命令を関数定義内で使用することはできない．

図 2.17 のように Unreal Script は，QuakeC においてアニメーションのコマごとに関数を記述するのと比較すると，ゲームキャラクタの「立ち」や「走り」等の状態をより記述しやすくなっている．

UnrealScript は 1.9 節の 4 つの要件について，次のように評価できる．

要件 1 State を使って状態遷移処理の記述が容易.

要件 2 Sleep 等の命令を使って時間に沿った処理の記述が容易.

要件 3 並行処理はキャラクタ単位でのみ可能.

要件 4 キャラクタ間の通信の記述に適した機能が無い.

[3]SPAWN の同名の State 機能とは異なる.

UnrealScript は，State 機能により状態遷移処理の記述が容易である．QuakeC と違い，1 つの状態を 1 つの `state` として記述できる，冗長な記述にもならない．

また，`Sleep` 等の一定時間処理を停止する命令を使い，時間に沿った処理の記述も可能である．ある状態内部の時間に沿った処理は，上から順にわかりやすく記述することができる．

UnrealScript では，自律して処理を進める Actor クラスをキャラクタとすることから，キャラクタ単位の並行処理が可能である．しかし，1 つのキャラクタに複数の並行処理を記述することはできない．

UnrealScript には，キャラクタ間の通信に適した記述機能は無い．そのため，キャラクタ間の通信をわかりやすく容易に記述することが難しい．

以上から，UnrealScript は 1.9 節の 4 つの要件を十分に満足していない．

2.4.4 対戦格闘ゲームの記述ツール

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，次の特徴がある．

- キャラクタは 2 人しか登場しない．
- 登場キャラクタは多くの小さなアニメーションを持つ．
- 複雑な状態遷移がある．
- 細かいパラメータの設定が必要である．

対戦格闘ゲームではゲーム内に登場するキャラクタは基本的に 2 体だけである．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．

格闘ゲームでは攻撃技や移動方法が多いほど遊び方のバリエーションが広がることから，キャラクタには多数の小さなアニメーションが用意される．キャラクタの動きはこの小さなアニメーションの組合せで表現される．

また，登場するキャラクタはプレーヤーの操作により，多くの技を連続で出したり，途中で変化させることができる．このような技を組み合せて使用するゲームシステムとすることでゲームに深みを出すことができる．

多彩な技とその組み合わせが格闘ゲームの主要な面白さとなることから，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) 等の技の属性の適切な設定が格闘ゲームの開発では重要な作業となる．

そのため，本節で紹介する記述ツールも，技の設定や，動作の記述が容易となる設計がされている．

```
/* 技 mh_n07_rp2 についての記述 */
MH(mh_n07_rp2)
  MH_HIGH(PUNCH_L, 0,12,0,35,11,
    ATF_NORMAL,TACHI,TACHI) /* 攻撃力，攻撃発生時間等の指定 */
  mh_yarare(HS,HD,HM,HD,DA, /* 攻撃成功時の相手のリアクション指定 */
    80,100*DEF_TOBI,
    90,120*DEF_TOBI,
    66,100*DEF_TOBI,
    100,100*DEF_TOBI,
    100,100*DEF_TOBI,
    100,100*DEF_TOBI)

  /* 技中の中段攻撃入力による遷移処理指定 */
  mh_cont_shift(MIDDLE_ATK, MN_M61_H3,13,32)

  /* 技中の上段攻撃入力による遷移処理指定 */
  mh_cont_shift(1F_HIGH_ATK, MN_N07_RP3,23,32)

  /* 技中の特殊な上段攻撃入力による遷移処理指定 */
  mh_cont_shift(JUST_HIGH_ATK, MN_M61_H2,13,26)

  pp_sound(10, sn_2_kaze02) /* 10 フレーム目に音を鳴らす指定 */
```

図 2.18: K1 の記述ツールでの記述例

格闘ゲーム K1

K1 は家庭用ゲーム機上で動作する対戦格闘ゲームである．ゲームデザイナーはテキストエディタ等を用いてテキストファイルを記述する．このテキストファイルは C 言語処理系に渡され，C 言語のマクロ機能によりゲーム A 用の C プログラムに変換され，ゲーム A プログラムの中に組み込まれる．

図 2.18 は K1 ゲームシステムの記述例である．K1 の記述ツールでの記述形式には次の特徴がある．

- 技単位で記述する．
- 様々なパラメータを指定する命令がある．
- 状態遷移命令がある．
- データ定義や制御構造文がない．

K1 の記述ツールの記述形式は，技の定義を記述の単位とする．**図 2.18** は先頭行にある mh_n07_rp2 という名前の技についての記述で，ゲームシステムの記述はこのような技の定義の集まりである．対戦格闘ゲームは 1 つの技がキャラクタの 1 つの状態と捉えると，キャラクタの動作をイメージしやすいので，この記述方法は格闘ゲームに適している．

各技の記述は，攻撃力や，技が成功時のリアクション種別等の，様々なパラメータを指定する命令の列挙で構成される．これらの命令はプログラマー

により組み込まれているもので，開発中に必要に応じてプログラマーによって追加，修正される．これらの命令の中には，キャラクタの状態やゲームのシチュエーションを条件として，状態 (技) の遷移を指定できるものもある．

記述は命令の列挙のみで，データ定義や制御構文などは存在しない．そのため，記述の自由度は低いが，理解しやすい構造となっている．

記述形式は命令の列挙のみで，データ定義や制御構造等の記述機能は無い．用意されていない新しい条件や処理の記述が必要な場合は，プログラマーが対応する新しい命令を実装しなければならない．

K1 の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 K1 のゲームシステムに適した状態遷移処理の記述が容易．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 並行処理の記述に適した機能が無い．

要件 4 キャラクタ間の通信の記述に適した機能が無い．

K1 の記述ツールは，K1 のゲームキャラクタの状態遷移処理を記述するのに適した機能を提供している．しかし，状態の遷移は，K1 のゲームシステムに依存した命令により記述しなければならず，実装されていない新しい条件の状態遷移を記述するには，プログラマーが対応する新しい命令を実装しなければならない．また，全ての命令は K1 のゲームに特化した設計となっており，他のゲーム開発に利用することが難しい．

また，条件や処理内容により異なる命令を用意しなければならず，時間に沿った処理の記述に適した機能があるとは言えない．

K1 ゲームは登場するキャラクタが 2 体だけで，開発ではキャラクタの種別ごとの状態遷移処理が記述できれば十分である．そのため，並行処理という概念が無く，並行処理に適した記述機能も無い．

キャラクタ間の通信についても，必要なものは必ずプログラマーが対応する命令を用意しなければならず，記述ツール自身に通信の記述に適した機能があるわけではない．

以上から，K1 の記述ツールは 1.9 節の 4 つの要件を十分には満足していない．

格闘ゲーム K2

K2 も K1 と同様に家庭用ゲーム機上で動作する対戦格闘ゲームである．**図 2.19** は K2 の記述ツールにおける記述例である．K1 同様に K2 の記述ツールもテキストファイルを入力データとする．K1 と比較して，字句や命令の名前等の，細かい記述ルールは違うものの，記述の構造は K1 と同様に，各技ごとに属性やパラメータを指定する命令を列挙するものであり，列挙する命令も類似したものが多い．

一方，K2 は次の点で K1 と異なる．

```
# H_JKD_STEP_LHK という技についての記述
[action]
  name "H_JKD_STEP_LHK"       # 技名
  motion "H_JKD_STEP_LHK" "" # アニメーションデータの指定
  shift button 39 39          # ボタン入力を受け付ける時間
  # 攻撃力や攻撃方向等の指定
  attack 18 asi_l -1 15 +1 hi none fr
  # 攻撃成功時の相手のリアクション指定
  yarare damage_hi_large -1 damage_hi_counter -1
  use_action_table Kamae_L   # この状態からの遷移の記述の指定

# Kamae_L という名前の付けられた状態遷移処理記述
[action_table]
  root Kamae_L "JKD_KAMA_LOOP_NEW_L"
    # ボタン a の時の遷移先指定
    file btn_a 0 "L_JKD_L_STEP_LOW_K"
      # ボタン a かつボタン y の時の遷移先指定
      file btn_y 0 "H_JKD_L_STEP_HK"
      end
    end
    # ボタン x の時の遷移先指定
    file btn_x 0 "M_JKD_L_SIDE_K"
    end
  end
```

図 2.19: K2 の記述ツール上での記述例

- 状態の処理と遷移の処理を別々に記述する．
- 遷移は階層化して記述できる．
- 記述したテキストデータをゲーム実行時に解釈する．

状態処理の記述は K1 と同じだが，状態遷移の処理については K1 と異なる．K1 のツールでは，状態の処理も遷移も列挙形式で区別せず記述するが，K2 のツールでは，遷移は状態の処理とは分け，列挙形式ではなく階層的に記述する．これにより「ジャンプボタンを押しながら攻撃ボタンを押した場合」等の，複雑な条件による遷移を記述しやすくなり，K1 の記述ツールよりも状態遷移の記述の自由度が高くなっている．

また，K1 の記述ツールが C 言語の前処理機能を利用したテキストの変換処理で動作するのに対して，K2 の記述ツールには特別な前処理プログラムは存在せず，ゲームプログラムが記述した文書ファイルをゲーム実行時に直接データとして読み込み解釈と実行を行う．解釈には専用のプログラムが用意されており，記述間違いの報告は K1 の記述ツールよりわかりやすい．

K2 の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 K2 のゲームシステムに特化した状態遷移処理の記述が容易．

要件 2 時間に沿った処理の記述に適した機能が無い．

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

各項目の説明
file 名: 使用するアニメーションデータ名の指定
種別: パンチやキック等の攻撃タイプの指定
判定: 攻撃判定が発生する時間の指定
判定モデル: 攻撃判定をする体の部位の指定
値: 攻撃力の数値を指定
攻撃方向: 正面，左右等の攻撃方向の指定
高さ: 攻撃の高さの指定

図 2.20: K3 の記述ツール上での記述例

要件 3 並行処理の記述に適した機能が無い．

要件 4 キャラクタ間の通信の記述に適した機能が無い．

K2 の記述ツールは，K1 の記述ツールと同様に，K2 のゲームキャラクタの状態遷移処理を記述するのに適した機能を提供している．しかし，K1 と同様，全ての命令は K2 のゲームに特化した設計となっており，ゲームデザイナーが独自に新しい条件や処理の命令を定義することができず，記述ツールを他のゲームシステムの開発に利用することも難しい．

必要な処理について，対応する命令を実装しなければならないことから，時間に沿った処理の記述に適した機能があるとは言えない．

K2 ゲームもキャラクタが 2 体のみ登場する格闘ゲームであることから，並行処理という概念は無く，並行処理に適した記述機能も無い．

全ての機能を命令として用意しなければならず，通信を記述するのに適した機能も無い．

以上から，K2 の記述ツールは 1.9 節の 4 つの要件を十分には満足していない．

格闘ゲーム K3

K3 は K1 および K2 と同じく対戦格闘ゲームである．K3 の記述ツールでも技が記述の基本単位となっているが，Excel[MCa] を用いて表データとして記述する点が K1 や K2 とは異なる．**図 2.20** は K3 の記述ツール上での記述例で，縦に技が並び，各セルには攻撃力や攻撃方向などの属性値を記述する．

K3 の記述ツールには次の特徴がある．

- 入力ツールに Excel を使用するので入力時点で記述間違いの報告ができる．

- Excel のデータは記述ツールによりゲーム専用のフォーマットに変換後，ゲーム実行時に解釈実行される．
- 縦に技，横に属性値という表形式の並びは，格闘ゲームの仕組みを理解している利用者にとって直感的に理解しやすい．

K3 の記述ツールは，入力ツールに Excel を使用するため，Excel の制約機能やマクロ機能などを用いることで，入力間違いをある程度事前に検査する機能や，入力可能な値を選択式とするインターフェースの追加をすることができる．

さらに，入力データは，記述ツールによりゲームソフトウェア用の専用形式に変換されるので，この時点でも記述間違いの検査が可能となっている．

多数の技をバランス良く調整することが格闘ゲームの面白さに繋がるため、技は見通しよく管理修正できることが望ましい．縦方向に技，横方向に属性値を並べる表形式の表現はこの点で優れた方法と考えられる．さらに K1 や K2 のようなテキスト形式の記述方式より表形式の表現は理解しやすく，追加や修正も容易である．

K3 の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 K3 のゲームシステムに特化した状態遷移処理の記述が容易．

要件 2 時間に沿った処理の記述に適した機能が無い．

要件 3 並行処理の記述に適した機能が無い．

要件 4 キャラクタ間の通信の記述に適した機能が無い．

K3 の記述ツールも，K1，K2 の記述ツールと同様に，K3 のゲームキャラクタの状態遷移処理を記述するのに適した機能を提供している．しかし，表の列ごとに記述可能な機能は決められており，K1，K2 同様，ゲームデザイナーが独自にゲームデザイナーが独自に新しい条件や処理の記述をすることはできない．

また，表形式であるため処理の流れを表現することは難しい．時間に沿った処理の記述をする機能も無い．

K3 ゲームもキャラクタが 2 体のみ登場する格闘ゲームであることから，並行処理という概念は無く，並行処理に適した記述機能は無い．

必要な処理は全て列として導入しなければならず，通信を記述するのに適した機能も無い．

以上から，K3 の記述ツールは 1.9 節の 4 つの要件を十分には満足していない．

2.4.5　複数キャラクタが登場するアクションゲームの記述ツール

対戦格闘ゲームと違い，この節で挙げるアクションゲーム A はゲーム中に同時にキャラクタが 3 体以上登場する．そのため，記述ツールも格闘ゲームのものと違い，より自由度の高い記述が可能なプログラミング言語を提供する．

```
/* 車キャラクタの記述 */
char Car : Obj {
 /* 回転速度と移動速度を指定する関数 */
 void set_rotvel(float rot, float vx, float vz) {
   set_rot([0, rot, 0]);
   set_vel([vx, 0, vz]);
 }
 /* 車キャラクタの記述 */
 @main() {
   load_poly("car_model");   /* 表示データ設定 */
   play_motion("drive_car"); /* アニメーションデータ設定 */
   set_pos([10, 0, 10]);     /* 位置設定 */
   set_rotvel(0, 0, 2);      /* 回転, 移動速度設定 */
  _D_WAIT(100);             /* 100 フレーム待つ */
   set_rotvel(45, 2, 2);     /* 回転, 移動速度修正 */
  _D_WAIT(50);              /* 50 フレーム待つ */
   set_rotvel(0, 0, 2);      /* 回転, 移動速度修正 */
  _D_WAIT(100);             /* 100 フレーム待つ */
   set_rotvel(-180, 0, -2);  /* 回転, 移動速度修正 */
  /* y 位置が-100 以下になるまで待つ */
  _D_WAIT_IF(get_pos().y < -100);
   hide();                   /* キャラクタを隠す */
   Bom.enable_bom();         /* 爆発キャラクタを起動する */
 }
}
/* 爆発キャラクタの記述 */
char Bom : Obj {
 /* 爆発開始を指示する関数 */
 int start;
 void enable_bom() { start = 1; }
 /* 爆発キャラクタの処理の記述 */
 @main() {
   start = 0;
  _D_WAIT_IF(start == 0);    /* 爆発開始が指定されるまで待つ */
   load_poly("bom_model");    /* 表示データ設定 */
   play_motion("effect_bom"); /* アニメーションデータ設定 */
  _D_WAIT(40);               /* 40 フレーム待つ */
   hide();                    /* キャラクタを隠す */
 }
}
```

図 2.21: A の記述ツール上での記述例

アクションゲーム A

ゲーム A はプレーヤーがキャラクタを操り，各シーンに登場する複数の敵をパンチやキックなどの技で倒していくゲームである．プレーヤーの味方となるキャラクタも存在し，それらのキャラクタと協力しながらゲームを進めていく．

A のゲームシステムの記述は，K1 や K2 と同様に，テキストエディタでテキストファイルを作ることで行う．テキストは記述ツールの処理により専用形式のデータに変換され，ゲームプログラムがこのデータを読み込み解釈及び実行をする．

図 2.21 は A の記述ツールでの記述例である．この例は，車と車の爆発の 2 つのキャラクタを記述している．A の記述ツールの記述形式は次の特徴を持つ．

- キャラクタは既存のオブジェクト指向言語と同じクラスとして記述する．
- 状態遷移処理の記述に適した機能がある．
- 制御構造を記述できる．
- 一定時間処理を停止する命令がある．
- 各クラスは一つのスレッドを持つ．

A の記述ツールの構文は C++言語 [Str98] や Java 言語 [Gos00] に近い．ゲーム中のキャラクタは，キャラクタに必要なデータと処理 (メソッド) をまとめたクラスによって記述する．データの定義や処理も，C++言語や Java 言語に類似した方法で記述をする．クラスは予約語 `class` で始まるが，`class` の代わりに `char` [4] を使用することでクラスで定義されるキャラクタをゲームシーン中に登場させることができる．C++言語や Java 言語にある継承機能も用意されている．**図 2.21** の例の先頭で `char Car : Obj` とあるのは，`Car` クラスが `Obj` クラスから継承していることを意味する．継承により親のクラスのデータやメソッドを再利用することが可能となり，効率的な開発が期待できる．

クラスの中には`@main` という名前の手続きが記述できる．この手続きにはキャラクタとして登場後，毎フレーム実行される処理を記述する．全ての登場キャラクタはこの処理が常に実行される．`@`で始まる他の名前の手続きを 1 つの状態として用意し，`_D_CHANGE(`名前`)` とすることで状態遷移の記述も可能である．

処理中は，**図 2.21** 中の `_D_WAIT(`フレーム数`)` という記述で，実行を一定時間停止させることができる．これによりゲーム中の時間の流れに沿った処理の記述が容易になっている．`if` や `while` 等の条件分岐やループ処理も用意されており，複雑な処理の記述が可能である．

また，クラス内で定義したデータやメソッドはクラス外の処理から参照，呼び出しが可能である．これを利用してキャラクタ間の通信を記述することができる．データのみで処理を含まない `char GLOBAL` というキャラクタを用意することで，グローバル変数の定義も可能である．

A の記述ツールは，前述の格闘ゲームのための記述ツールと違い，より汎用的なプログラミング言語に近い設計の記述言語を提供している．そのため，ゲーム A のアクション本編のゲームシステム記述にとどまらず，ゲームタイトル画面，キャラクタ選択モード，ストーリー描写シーン等の記述や，異なるルールで遊べるミニゲームなどの記述にも用いられている．

A の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述が容易．

要件 2 `_D_WAIT` 命令を用いて，時間に沿った処理の記述が容易．

[4] 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予約語である．

要件 3 キャラクタ単位の並行処理の記述が可能.

要件 4 キャラクタ間の通信の記述に適した機能は無い.

A の記述ツールでは，@で始まる名前の手続きと，_D_WAIT 命令を用いて状態遷移処理の記述が容易である.

また，A の記述ツールでは，_D_WAIT 命令により処理を指定したフレーム数だけ停止させることができ，時間に沿った処理の記述も容易となっている.

A の記述ツールでは，クラス単位での並行処理の記述が可能である．しかし，1 つのクラスが複数の並行処理を持つことはできない.

A の記述ツールには，キャラクタ間の通信を記述するのに便利な機能も用意されていない．グローバル変数やメソッド呼び出しを利用できるが，キャラクタを特定せずに情報を共有することは可能でも，1.9 節で例として挙げた通信の記述には適していない.

以上から，A の記述ツールは 1.9 節の 4 つの要件を十分には満足していない.

2.4.6 シューティングゲームの記述ツール

シューティングゲームは，プレーヤーが弾を敵に向かって撃ち敵を倒すゲームである．敵や弾は出現と消滅を繰り返すことから，記述システムには生成と消滅の機能が必要となる．また，様々なキャラクタ同士の衝突判定や，編隊を組んだ攻撃などの処理が必要なことから，キャラクタ間に発生するイベント処理や，キャラクタの情報をやり取りする機能も必要とされる.

シューティングゲーム S1

S1 は携帯機器用の画面が縦にスクロールするタイプのシューティングゲームである．記述ツールは自機のパワーアップの設定や敵機の移動，出現処理の記述に用いられている.

S1 の記述ツールでの記述は K1 や K2 と同様にテキストファイルとして記述することで行う．テキストファイルは記述ツールの処理プログラムにより Java 言語のプログラムへ変換され，アプリケーション本体の Java プログラムと結合される.

図 2.22 は S1 の記述ツールでの記述例である．この例は，画面内を移動しながら繰り返し弾を撃つキャラクタを記述している．この記述ツールでの記述形式には次の特徴がある.

- キャラクタ毎に処理を記述する.
- 制御構造を記述できる.
- 一定時間処理を停止する命令がある.
- 1 つのキャラクタに複数の並行する処理を記述できる.

```
enemy ENEMY
  pos swidth/2, top      # 初期位置設定
  dir 90deg              # 初期方向設定
  speed 2.0              # 初期移動速さ設定
* 100                    # 100 フレーム待つ
  turn left 45deg        # 45 度左へ向きを変える
* 50                     # 50 フレーム待つ
  turn right 45deg       # 45 度右へ向きを変える
* 100                    # 100 フレーム待つ
  dir up                 # 上方向を向く
* after
  dead if out            # 画面外に出たら処理終了
* killed
  create BOM :pos x, y   # 殺された場合 BOM を作る
-----------------
# 40 フレーム待ってループ処理開始
# ループ中は 30 フレームおきに TAMA を作る
* 40 loop
  create TAMA :pos x, y
* 30 endloop
end
```

図 2.22: S1 の記述ツールでの記述例

- 特定のイベントに対応する処理を記述できる．
- グローバル変数を利用できる．

S1 の記述ツールではキャラクタ毎に処理を記述する．処理中では if，while 等の制御構造を記述することができる．goto の記述も可能である．

処理中で「* 100」と記述することで 100 フレームの間，処理を停止することができる．この機能により，ゲーム中の時間の流れに合わせた処理の記述を容易にできる．

1 つのキャラクタ中に複数の並行する処理の記述も容易である．**図 2.22** の例では，移動処理と弾射出処理を別々の並行する処理として記述している．

記述の中では**図 2.22** にある「* after」や「* killed」という記述を用いて，特定のイベントに対応する処理を記述可能である.「* after」以降には先行するすべての処理終了後の処理を,「* killed」以降にはキャラクタが破壊されたときの処理を記述する．このほかにダメージを受けたときの「* damaged」などがある．

また，グローバル変数が使え，キャラクタ全体で情報を共有することに利用できる．

S1 の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能は無い．

要件 2 「* 数値」記述により，時間に沿った処理の記述が容易である．

要件 3 並行処理の記述に適した機能がある．

要件 4 キャラクタ間の通信の記述に適した機能は無い.

S1 の記述ツールには，状態遷移処理を記述するのに適した機能はない．`goto` や `if` 等の制御構造を用いて記述することは可能だが，見通しが悪く，間違いの原因となりやすい記述になる．

S1 の記述ツールでは,「`*` 数値」と記述することで処理を一定時間停止することができ，時間に沿った処理の記述が容易になっている．

S1 の記述ツールは，並行処理の記述に適した機能を持つ．キャラクタは自律的に処理を進めるようになっており，1 つのキャラクタに複数の並行処理を記述することもでき，

キャラクタ間の通信を記述するのに適した機能は用意されていない．グローバル変数を用いてキャラクタを特定せずに情報を共有することは可能だが，ゲーム中の様々な通信を記述するのは難しい．`*killed` 等のあらかじめ用意されたイベントに反応する処理を記述できるが，それ以外の処理を`*`で記述することはできない．

以上から，S1 の記述ツールは 1.9 節の 4 つの要件を十分には満足していない．

シューティングゲーム S2

S2 はプレーヤーキャラクタを操り自機の撃つ弾で敵を倒していくゲームである．記述ツールはゲームシステムやデモンストレーションシーンの記述に用いられている．S2 の記述ツールでは K1 や S1 と同様テキストファイルとしてゲームシステムを記述する．記述したテキストファイルは記述ツールの処理により専用形式のデータに変換され，ゲーム実行時に解釈実行される．

図 2.23 は S2 の記述ツールでの記述例である．この例は，車キャラクタの移動状態，攻撃によりダメージを受けたときの状態，クラッシュするときの状態の 3 つの状態についての記述である．この記述ツールには次の特徴がある．

- キャラクタの状態を記述の単位とし，状態遷移の記述ができる．
- 制御構造の記述ができる．
- 一定時間処理を停止する命令がある．
- 特定のイベントに対応する状態を指定できる．
- 処理中でゲーム中の情報にアクセスすることが可能．
- デバッグ機能が用意されている．

S2 の記述はキャラクタの状態を 1 つの手続として，その組み合わせによりキャラクタの動作を記述する．状態は記述者が任意に変更可能で，状態遷移処理の記述がしやすい．状態変更後，処理が終了したら元の状態に戻ってくる処理の記述もできる．状態の変更は処理対象となるキャラクタだけでなく，

```
/* 車キャラクタの処理の定義 */
car1 () {
 hit_point ( 5 );              /* 耐久力 5 */
 model(MODEL_CAR_TAXI);        /* 表示データ設定 */
 suffer_svsection (car1_hurt);   /* ダメージ時の処理指定 */
 dying_svsection (car1_crash);   /* 死んだときの処理指定 */
 set_posi(-30m, 0, 10m);       /* キャラクタ位置設定 */
 set_roll(0, 0x4000, 0);       /* キャラクタの向き設定 */
 set_move_mode(
    MOVE_AIM_STOP,
    MOVE_AIM_STOP);            /* 移動モード */
 move_posi(8:00, 10m, 0, 10m); /* キャラクタの向き指定 */
 wait(100);                    /* 100 フレーム待つ */
 set_roll(0, 0x0000, 0);       /* キャラクタの向き指定 */
 set_speed(0, 0, 2);           /* キャラクタの速度指定 */
 wait(50);                     /* 50 フレーム待つ */
 set_roll(0, 0x4000, 0);       /* キャラクタの向き指定 */
 set_speed(0, 2, 2);           /* キャラクタの速度指定 */
 wait(100);                    /* 100 フレーム待つ */
 set_roll(0, 0x0000, 0);       /* キャラクタの向き指定 */
 set_speed(0, 0, 2);           /* キャラクタの速度指定 */
}
/* ダメージ時の処理の定義 */
car1_hurt () {
 move_roll(8, 0x400, 0x4100, 0);   /* 向きを少し変える */
 wait 8;                           /* 8 フレーム待つ */
 move_roll(16, -0x800, 0x3f00, 0); /* 向きを少し変える */
 wait 8;                           /* 8 フレーム待つ */
}
/* 死んだときの処理の定義 */
car1_crash () {
 hit_point(-1); /* 無敵にしておく (2 度以上死なないようにする) */
 move_posi_rel(3:00, 0, 0, 2m);
 move_roll_rel(1:00, 0, 0x2000, 0);  /* がたがたさせる処理 */
 wait 5:00;                          /* 5 秒待つ */
}
```

図 2.23: S2 の記述ツールでの記述例

処理対象外のキャラクタに対しても可能で，4 つまで引数を伴うこともできるので，通信機能として利用することが可能である．

状態処理には制御構造を使用できるので，状態内部での複雑な処理やループ処理ができる．

A や S1 と同様，S2 の記述形式にも処理を停止する命令 wait が用意されている．S2 の場合，A や S1 と違い待つ時間としてフレームを指定できるだけでなく，秒による指定をすることもできる．

S1 のような特別なイベントのための記述は S2 にも用意されている．**図 2.23** の記述中の「suffer_svsection()」や「dying_svsection()」という記述は，それぞれ他のキャラクタによる攻撃で傷ついたときと，キャラクタが破壊されたときに遷移する状態を指定している．

S2 は S1 と比較して記述の自由度が高い．また，記述のための環境も整備されている．たとえば，実行時デバッグ用のツール等も用意されており，実行時に記述したプログラムを修正して再実行することも可能である．

S2 の記述ツールは，1.9 節の 4 つの要件について，次のように評価できる．

要件 1 状態遷移処理の記述に適した機能を持つ．

要件 2 `wait` 命令により，時間に沿った処理の記述が容易である．

要件 3 キャラクタ単位の並行処理記述が可能である．

要件 4 キャラクタ間の通信に，状態を変更する機能が利用できるが，限定された場合にしか利用できない．

S2 の記述ツールの記述形式は，1 つの状態を 1 つの手続きとして記述でき，キャラクタの状態遷移処理の記述に適した設計となっている．

S2 の記述ツールには，一定時間処理を停止する `wait` 命令が用意されている．これにより，時間に沿った処理の記述が容易になっている．

S2 の記述ツールでは，キャラクタは自律して処理を進める．しかし，1 つのキャラクタに複数の並行処理を記述することはできない．

S2 の記述ツールには，指定のキャラクタの状態を変更する機能がある．しかし，この機能は 1 対 1 の通信で，1.9 節で述べた，1 対 1 ではない通信の記述には適さない．

以上から，S2 の記述ツールは 1.9 節の 4 つの要件を十分には満足していない．

2.4.7 個々のゲーム開発専用記述ツールのまとめ

本節では，個々のゲーム記述専用に開発された記述ツールについて述べた．**表 2.4** は説明した記述ツールについてまとめたものである．

1.9 節で述べたように，4 要件は，ゲームデザイナーがゲームシステムを記述することを容易にするために仮定した要件であり，要件を満たさなくともゲームシステムの記述は可能である．しかし，ゲームシステムの実装が進み，規模が大きく複雑になると，要件を満たさない言語では可読性が悪くなり，修正が容易ではなく，トライアンドエラーの難しい記述になる．

本節で述べた記述ツールは，個々のゲームに特化した設計や実装となっていることから，一般的なプログラミング言語と違い非常に独特な文法や記法を持つ．

特に非公開の記述ツールは，他のプログラミング言語や記述ツールと比較しても特徴のある文法や形式となっている．これは，公開されているツールが，ユーザーに利用されることを考慮して，理解や学習の容易な既存の言語 (C や C++等) に近いものとしているのに対し，非公開のものは，開発内部で一部のスタッフが利用するのみであることから，出来る限り目的ゲームに適した設計にしたことが理由であると考えられる．

表 2.4: 個々のゲーム専用の記述ツールの比較

言語名	ゲーム名	ゲームのタイプ	プラットフォーム	状態遷移	時間	並行処理	通信
QuakeC	Quake	FPS	Windows	△	△	△	×
Unreal Script	Unreal Tournament	FPS	Windows	○	○	△	×
	K1	対戦格闘	家庭用ゲーム機	※	※	×	×
	K2	対戦格闘	家庭用ゲーム機	※	※	×	×
	K3	対戦格闘	家庭用ゲーム機	※	※	×	×
	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	シューティング	業務用機	○	○	△	△

○ 十分満たしている.
△ 機能として提供されているが十分ではない.
× 機能が提供されていない.
※ 目的ゲームに特化しすぎているもの.

本節で述べたツールは，目的ゲームに特化しており，他のゲームへの再利用性はあまり考慮されていない．例えば，K1，K2，K3 の格闘ゲームの記述ツールは，技のバリエーションや細かい数値調整には適しているが，シューティングゲーム等の他の種類のゲームの記述は不可能である．その他の記述ツールも，目的とするゲームのみカスタマイズが可能な設計となっている．特定の要素の記述だけ可能な記述ツールは，それ以外の記述を難しくし，異なるルールのゲームシステムの記述には不向きであったり，トライアンドエラーの範囲を限定する要因となる．

これらのことから，本節で述べた記述ツールは，1 章で述べた要件を十分には満たしていない．

2.5　他分野のアプリケーション開発ツール

2.5.1　他分野のアプリケーション開発ツールの位置づけ

ビデオゲームそのものは目的としていないが，類似したアプリケーションの開発を目的としたツールが多く存在する．

2.5 節で説明した ALICE[CAB+00] や Obliq-3D[NB95] 等のアニメーションオーサリングツールは，3D アニメーションの開発を目的としたソフトウェアである．これらはビデオゲームの開発を目的としたソフトウェアではないが，アニメーション動作をするオブジェクトの記述や，アニメーション上のオブジェクト間の関連を記述することもできることから，ビデオゲームシステムの記述に適していると考えられる．

また，AgentSheets[Ale93]，StarLogo[斎藤 09]，AScape[Par01] は，社会現象をシミュレーションするために使われるエージェントシミュレーションツールで，アニメーションオーサリングツール同様にビデオゲーム開発を目的と

していない．しかし，自律して動作するエージェントとエージェント間の関連を記述することでシミュレーションを行う点が，ビデオゲームシステムに類似しており，簡単なビデオゲームを実装することが可能である．

プログラミング教育を目的としたツールには，プログラミン [文部 10] や，ドリトル [兼宗 01] のように，プログラミングに興味を引かせるために簡単なゲームを作ることが可能なものもある．

本節では，代表的なものとして，ALICE，AgentSheets，及びプログラミンについて説明する．また，最後の 2.5.6 節において，これらのツールについての評価を行う．

2.5.2 ALICE

ALICE[CAB+00] は 3D グラフィクスのアニメーションを製作するためのオーサリングツールである (**図 2.24**)．ALICE は，プログラミングの経験がないユーザーであっても，3D グラフィクスアニメーションを簡単に作成できることを目的としたツールである．ALICE を用いることで，複数のキャラクタを組み合わせた複雑なアニメーションを，比較的簡単に記述することができる．ALICE には次の特徴がある．

- 制作作業はすべて GUI 上で行う．
- アニメーション中のオブジェクトを階層的に管理できる．
- 自律的なアニメーションのために並行処理が記述しやすい．

ALICE の制作作業はすべて GUI 上で行い，アニメーション中に登場するキャラクタの動作は，ビジュアル言語を利用して記述する．例えば，**図 2.24** はアニメーション画面中のペンギンキャラクタのアニメーションを編集中の画面で，右下のペインには，そのペンギンの動作が矩形の階層構造で記述されている．矩形は上から下へ順番に処理される．矩形には「指定方向へ動かす」「しばらく待つ」といった単体で動作するものや,「同時に実行する」や「指定回数繰り返す」などの制御構造を表すものがあり，これらを組み合わせることで様々な動作を記述可能である．

また，3D グラフィクスのアニメーションを制作する場合，シーン中のオブジェクト間に親子関係を持たせると管理がしやすい．ALICE でもすべてのオブジェクトはこの階層構造の中で管理される．**図 2.24** の左側には，この階層構造が表示されている．

オブジェクトは自律的にアニメーション処理を実行するので，並行処理の記述と時間の流れに沿った記述もしやすくなっている．全てのキャラクタは処理を同時に実行し，動作の記述に利用する「同時に実行する」矩形により，1 つのキャラクタ内部でも複数の並行する処理を記述可能である．

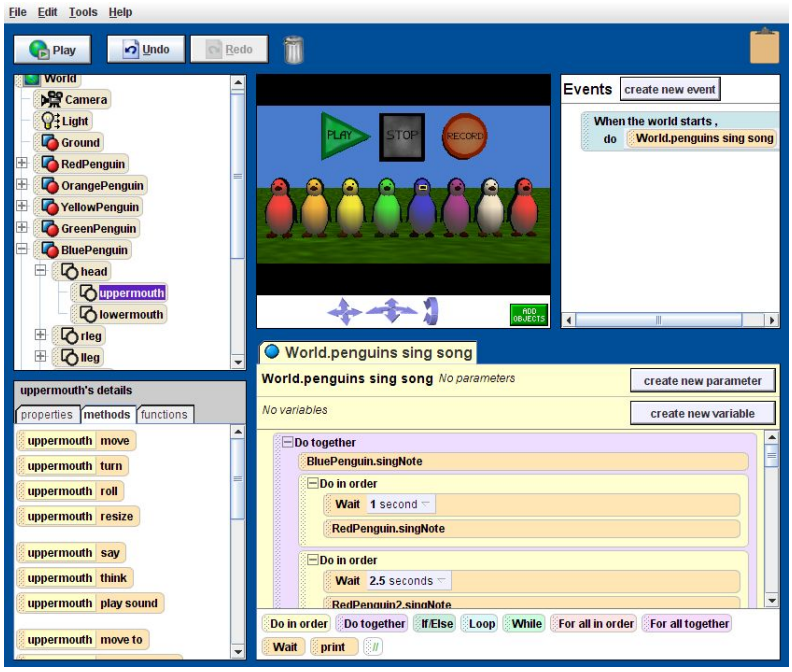

図 2.24: Alice の画面

2.5.3　AgentSheets

エージェントシミュレーションツールは，自律して動作するエージェントを用いて，特定の社会現象のシミュレーションや，モデル化を行うためのツールである [斎藤 09, Par01]．AgentSheets[Ale93] もエージェントシミュレーションツールで広く用いられている．AgentSheets には次の特徴がある．

- 制作作業はすべて GUI 上で行う．
- エージェントの動作は IF-THEN ルールにより記述する．
- エージェントはルールに従って自律して動作する．

AgentSheets は制作作業はすべて GUI 上で行い，比較的簡単にエージェントシミュレーションの記述が可能になっている．

エージェントの動作は IF-THEN ルールの集まりで表現される．IF-THEN ルールに与えられる条件中では，シミュレーションシーンの情報や，近くのエージェントの情報を参照することが可能なので，エージェント間の通信を記述することができる．

エージェントは与えられる IF-THEN ルールに従って自律的に動作する．この点はビデオゲームのキャラクタの動作に近く，AgentSheets を利用して，簡易なゲーム制作も可能である．

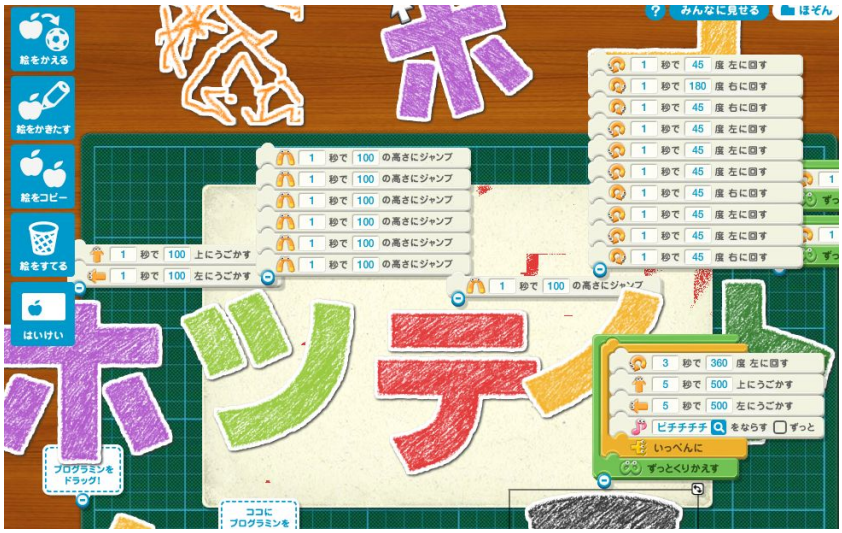

図 2.25: プログラミンの画面

AgentsSheets のインターフェースは比較的シンプルで分かりやすく，アニメーションや簡単なビデオゲームを製作することも可能なことから，プログラミングの教育用としても用いられている．

2.5.4 プログラミン

プログラミン [文部 10] は，子供がプログラミングを簡単に実践できることを目的とした Web ブラウザ上で利用可能な，ビジュアルプログラミング環境 (**図 2.25**) である．

プログラミンも Alice や AgentsSheets のように，シンプルでわかりやすいインターフェースを持ち，手軽に簡単なアニメーションやインタラクティブなアプリケーションを制作することが可能である．

記述言語はビジュアル言語で，各表示キャラクタごとに，下から上へ積み上げるような形で処理を表す矩形を列挙する．プログラムは積み上げた矩形を下から上へ順番に実行する．この記述方法は，処理順序が逆であること以外 Alice によく似ている．

矩形の中には，マウスやキー入力を待つものや，内包する矩形の列を繰り返したり，複数の矩形を同時に実行するものもある．例えば，**図 2.25** の右下は 4 つの処理を「いっぺんに」で同時に実行し，その処理を「ずっとくりかえす」で無限に繰り返すプログラムとなっている．この機能も Alice によく似ている．

これらを組み合わせることで，プログラミンでは簡単なゲームアプリケーションの記述も可能である．

2.5.5　その他の開発ツール

その他にも，ビデオゲームに類似したアプリケーションの開発ツールがある．

Lego Mindstorm[LEG98] は，動きをプログラマブルに変更可能なトイブロックで，動作の記述には専用のビジュアル言語を使用する．

Squeak eToys[BPT10] は，Smalltalk(Squeak) 環境上に開発されたビジュアルプログラミング環境である．タイルスクリプトと呼ばれる独自のビジュアル記述言語を用い，Morph という画面上のパーツに対して，ドラッグ&ドロップや簡単なマウス操作で，アニメーション等の記述が可能である．マウス操作だけでは難しい詳細な定義は，Smalltalk 言語を使い記述することもできる．Squeak eToys は，プログラミングの教育用の教材としても用いられている．

Squeak 環境上に開発された教育用のプログラミング言語として，Scratch[MBK+04] も挙げられる．Scratch は，Alice に似たブロックを組み合わせる方法でプログラムを記述する．

2.5.6　他分野のアプリケーション開発ツールのまとめ

本節では，ビデオゲームに類似するアプリケーションの記述が目的のツールについて説明した．これらは，ビデオゲームそのものを目的とはしていないが，ビデオゲームに似たアプリケーションを比較的容易に開発できることについて説明した．

どのツールも，前章までに述べたプログラミング言語や記述ツールと異なり，わかりやすいインターフェースで，制作も容易である．記述は，矩形やブロック等を並べたりつなげたりするものもあり，規模が小さければ，見通しが良く直感的にも理解しやすい．

本節で述べたツールは，1 章で述べたゲームシステムの記述の要件である，時間の流れに沿った処理や並行処理を記述する機能を持つ．

しかし，これらのツールは，状態遷移処理やキャラクタ間の通信を記述する機能を持つものが無く，ゲームシステムの記述に十分に適しているとは言えない．

また，矩形やブロックを並べて記述する方法は，プログラムが大きくなると見通しが悪くなり，修正や追加も困難になる．複雑なルールを記述しなければならないゲームシステムの開発では，この点も問題となる．

2.6 ゲーム記述に適した通信モデル

2.6.1 ゲームシステム記述と通信モデル

本章で説明した既存の記述ツールには，キャラクタの状態遷移や時間の流れに沿った処理及び並行処理を，平易に記述可能なものが少なくない．一方で，ゲームシステムのキャラクタ間の通信については，1 対 1 等特定の場合にのみ有効な機能であったり，グローバル変数や手続き呼出を利用する汎用的なプログラミングによる記述しかできない等，適切な機能が提供されているものが無い．

ビデオゲームのキャラクタが自律的に処理を進める仕組みは，並行/並列処理を基本として実装されるプログラムに近い．並行/並列処理を行うプログラムの開発方法については，計算機科学分野において古くから研究され，並行/並列処理間の通信を表現するためのモデルの提案も行われており，それらのモデルを実装した通信機能を持つプログラミング言語も多い．

そこで，計算機科学分野において研究されている通信モデルについて，キャラクタ間の通信を表現する方法として適切かどうかの調査を行った [NK11b]．

ゲームキャラクタ間の通信を表現するための通信モデルは，次の性質を満たすことが望ましい．

- 理解しやすく通信が簡潔に記述できること．
- 複数の並行処理間の通信の記述に適していること．
- キャラクタの種類や状態に依存した処理が記述に適していること．
- 処理順序やタイミングを明示できること．

トライアンドエラーを効率よく行うためには，ゲームデザイナーがプログラマーの協力無しにゲームシステムを記述できることが望ましい．ゲームデザイナーはプログラミングに精通していないことから，記述に使用するプログラミング言語や通信モデルは，ゲームデザイナーが理解できる，わかりやすいものが望まれる．また，冗長で複雑な記述は間違いの原因となりやすいことから，簡潔に通信を記述できるものが良い．

ゲームキャラクタは通信しながらも自律して動作を進める．例えば，プレーヤーキャラクタの攻撃が敵キャラクタに当たった場合，プレーヤーキャラクタは攻撃後のアクションを進め，敵キャラクタは攻撃によるダメージを受けるアクションを進める．よって，プログラミング言語には，並行処理間の通信の記述に適した機能が必要である．また,「ボスが死んだら子供も死ぬ」「全敵キャラクタにダメージ」「プレーヤーに向かって複数の敵キャラクタ組織立って攻撃」のような通信は 1 対 1 の通信ではない．よって，通信モデルは，複数のキャラクタを対象とした通信の記述に適したものが良い．

また，キャラクタ同士の関係は,「プレーヤーはロープに捕まって移動できる」「ガード中は攻撃は無効」「プレーヤーキャラクタは無敵状態中は敵キャ

ラクタに体当たり可能」のように，キャラクタの種別や状態に依存した処理であることも多い．よって，キャラクタの種類や状態に依存した通信も記述できる必要がある．

衝突処理を必要とするゲームでは，同時に複数の衝突が発生した場合に衝突の処理順序を決めておきたいことが多い．これは，同時衝突の処理順序が曖昧だとプレーヤーにとってゲームルールが曖昧に感じられるからである．また,「ゲームの目的達成でクリア画面」と「敵キャラクタに倒されゲーム終了画面」のような相反するイベントが同時発生した場合は，必ずどちらかだけを処理しなければならない．これらのことから，通信の処理順序やタイミングは明示できなければならない．

次節からは，これらの要件を基に，調査した通信モデルの特徴について説明し，それらがキャラクタ間の通信の記述に適しているかの検討を行う．また，最後の 2.6.10 節にて，調査した通信モデルについてまとめを行う．

2.6.2　手続き呼び出しと RPC

手続き呼び出しは，特定の処理を行う命令列を 1 つの「手続き」としてまとめ，その処理を必要とする側が手続きを呼び出す計算モデルである．手続き呼び出しは多くのプログラミング言語で広く用いられている．

手続き呼び出しは，命令列を手続きとしてまとめて必要な場所でその手続を呼び出すことで，プログラム中に同一の処理を何度も記述することを避け，プログラムの可読性や簡潔性を維持することができる．また，手続きを呼び出す際，処理に使用するデータを引数として渡し，同一の処理を異なるデータで実行させることもできる．手続き呼び出しは，スタック等の呼び出し側と呼び出される手続き側の双方で共有可能な記憶領域を利用して実装される．

手続き呼び出しを異なるプロセスやアドレス空間の間でも可能にしたのが，RPC(Remote Procedure Call) である．プロセスやアドレス空間が異なると手続き呼び出しのように双方で共有できる記憶領域が無いことから，RPC では手続き名や引数等を組としたメッセージというデータをやりとりするメッセージパッシングと呼ばれる手法で実装される．

手続き呼び出しや RPC の処理の流れを**図 2.26** に示す．呼び出し側が手続きを呼び出すと，処理の流れは手続き側に渡り，呼び出し側の処理は中断する．手続きの処理が終わると，手続きの結果とともに処理の流れが呼び出し側に戻り，処理が再開する．このように，手続き呼び出しは簡潔で理解しやすく，RPC のように並行する処理間での実装も可能である．また，処理の順序は呼び出し順なので，理解しやすく，処理順も明示しやすい．

手続き呼び出しは，多くのプログラミング言語で用意されている機能であることから，ゲームプログラミングやゲームシステム記述ツールでもキャラクタ間の通信を記述する方法として広く利用されている．1.9 節のキャラクタ間の通信でとりあげた Java 言語プログラムはその例である．2.2 節で説明し

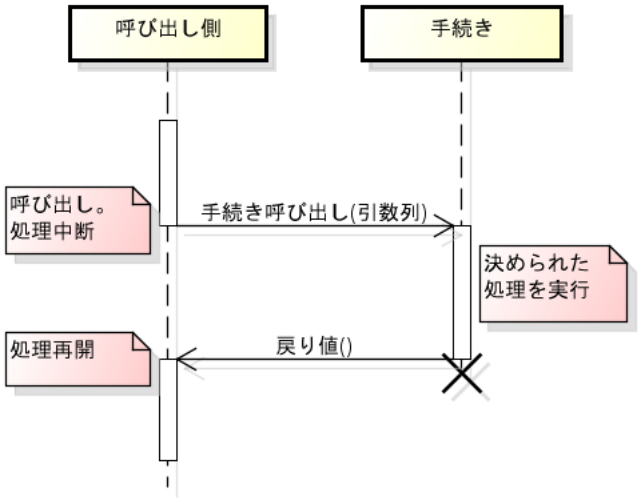

図 2.26: 手続き呼び出しの処理の流れ

たプログラミング言語も全て手続き呼び出し機能を提供している．2.3 節の Unity や Action Script も手続き呼び出しを使って通信の記述が可能である．

しかし，手続き呼び出しは，呼び出し側の処理が，手続きの処理が終了するまで停止する排他的な処理構造のため，並行する処理が通信しながら実行を進めるような場合に適していない．また，呼び出す側は任意の手続きを呼び出せるが，呼び出される手続きは呼び出し元へしか処理や結果を返すことができない．この階層的な関係は，自律した処理同士の通信としては適切ではなく，手続き呼び出しを使用してキャラクタ間の通信を記述すると，処理の流れが複雑になり，可読性の悪いプログラムとなりやすい．よって，相互に関係するビデオゲームのキャラクタ間の通信モデルとしては不適切である．

2.6.3 コルーチン

コルーチンは，手続き中でも処理の中断や再開を可能にすることで，協調して並行動作する処理を記述できるようにした計算モデルである．

コルーチンの処理の流れを**図 2.27** に示す．コルーチンを採用しているプログラミング言語では，手続きの定義中で処理を中断して結果を返す命令を記述できる．手続きが呼び出されその命令まで実行が進むと，処理は中断し呼び出され側に処理が戻るが，再度その手続が呼び出されると，実行は中断したところから再開する．

次は，本章で説明した Lua のコルーチンを利用した簡単な例である．

```
function sub_a( )
  coroutine.yield( 1 ) -- 1 を返して中断
  coroutine.yield( 2 ) -- 2 を返して中断
```

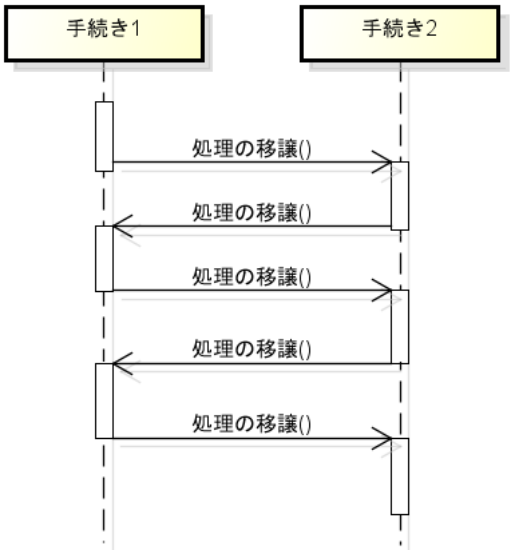

図 2.27: コルーチンの処理の流れ

```
  coroutine.yield( 3 ) -- 3 を返して中断
end

local call_a = coroutine.wrap( sub_a )
print( call_a( ) )  -- sub_a を呼び出して値を表示
print( call_a( ) )  -- sub_a を呼び出して値を表示
print( call_a( ) )  -- sub_a を呼び出して値を表示
```

この例では，sub_a は呼び出し側と並行して処理を進める．呼び出し側が sub_a を呼び出すたびに，sub_a は処理を少しづつ進めて異なる値を返す．このように，コルーチンを使うと協調して動作する並行処理を記述することができる．

コルーチンは，Lua や Squirrel，Stackless Python 等，2.2 節で述べたビデオゲーム開発を目的としたプログラミング言語でも提供されている．コルーチンを使ったゲームプログラミング手法も知られている [Daw02b]．

しかしコルーチンは，手続き呼び出しと同じく，呼び出す側は任意の手続きを呼び出せても，呼び出される側は処理を中断して呼び出し元へ戻る事しかできず，多数の並行する処理の間で多様な通信を記述するのには適していない．したがって，キャラクタが自立して処理を進めながら，複数のキャラクタと様々な通信を行うビデオゲームのキャラクタ間の通信モデルとしては不十分である．

また，コルーチンでは，処理が呼び出す側と呼び出される側を往復するので，多くの並行する処理がある場合，複雑で理解の難しいものとなりやすく，ゲームデザイナーがゲームキャラクタの挙動の記述のために使用するのには適していない．

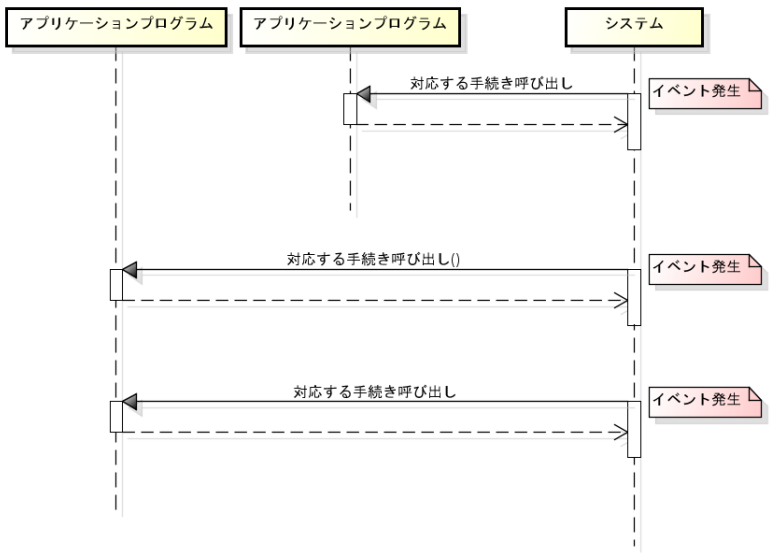

図 2.28: イベント駆動型プログラミングの処理の流れ

2.6.4 イベント駆動型プログラミング

イベント駆動型プログラミングは，イベントというシステムや他のプログラムの変化をアプリケーションに通知し，アプリケーションはその通知に応じた処理を行うプログラミングモデルで，多くのシステムで採用されている [BK09, win01, Har99].

イベント駆動型プログラミングを提供するシステムでは，イベントを処理するプログラムとして手続きを使用することが多い．システムを使用するアプリケーションプログラムは，イベントに応じた処理を実行するイベントハンドラと呼ばれる手続きを，システムにイベントと関連付けて登録する．システムは，この関連付けられた手続きをイベント発生時に呼びだすことで，アプリケーションプログラムのイベントに応じた処理を実行する (**図 2.28**). 手続きとイベントの関連付けは手続き名等を利用した規約として決められていることもある．

また，イベントをメッセージとして明示的に送信し，メッセージを待ち受ける命令等を用意して，アプリケーションプログラム側でメッセージを待ち受ける実装方法もある [win01].

イベント駆動型プログラミングは，手続き呼び出しと違い，アプリケーションプログラムに手続きを呼び出す処理の記述は必要なく，イベントに対応する処理のみ記述すればよい．そのため，見通しがよく，プログラムの可読性が良くなる．

イベント駆動型プログラミングは，ゲームプログラミングやゲームシステム記述ツールでも広く採用されている．例えば，2.2.3 節の Pawn の Event や，2.2.7 節で説明した Uncharted2 の Scheme のシグナルは，イベント駆動型プ

ログラミングをモデルとした通信記述機能である．

Uncharted2 の Scheme では，`signal` により，イベントを発生させることもできる．2.2.6 節で説明した Stackless Python のチャネルも，チャネルの `send()` メソッドを用いてイベントをメッセージとして送信し，`receive()` メソッドでイベントを待ち受ける処理の記述ができる．

2.3.2 節で述べた Unity の，毎フレーム呼び出される関数 `Update` や，他のキャラクタとの衝突時に呼び出される `OnControllerColliderHit` も，フレーム毎もしくは衝突時に発生するイベントに対応したイベントハンドラである．

その他，ゲームキャラクタを管理するシステムの機能の一つとして提供されたり [Har05]，非プレーヤーキャラクタの挙動処理プログラム (AI [5]) で，ゲームシーンの変化を挙動処理プログラムで処理するための実装方法としても利用されている [Rab04]．

イベント駆動型プログラミングは，予め決められたイベントに対する手続きを記述するだけで良いことから，理解しやすい．イベントの処理は，キャラクタの自律した処理とは異なるイベントハンドラが実行するので，手続き呼び出しのように，処理の流れが複雑になることもなく，簡潔性を維持しやすい．

しかし，イベント駆動型プログラミングでは，システム側で用意しているイベントの処理しか記述できないので，Uncharted2 の Scheme のようにイベントを発生させる機能が無ければ，システムが用意しないゲームシステム固有のキャラクタ間の通信に使用することが難しい．イベントはあらかじめ決められた条件で発生するので，キャラクタの種類や状態に応じたイベント発生や処理の記述も難しい．さらに，イベントの処理タイミングを記述する方法は無く，通信の処理順序やタイミングを明示することはできない．

2.6.5　データバインディング

データバインディング [SMK09] は互いに関連付けられたプログラム要素が，変化を相手に通知することで要素の更新を行うデータ同期モデルである．

データバインディングの例として Excel[MCa] 等の表計算アプリケーションが挙げられる．この種のアプリケーションでは，あるデータセルを，元になるセルの 2 倍の値となるよう設定すると，元のセルの値が変化する度に，設定されたセルもその 2 倍の値となるように更新される．また，ActionScript[SR08] では，変数にユーザーインターフェース上の要素を関連付けることができる．これにより，変数に値を設定するとユーザーインターフェース上の値が変化し，逆にユーザーインターフェース上で値を入力するとその値が変数に反映されるという動作の記述が容易となっている．

[5] ビデオゲームプログラミングにおいて，プレーヤーに対する敵キャラクタのような，ゲーム状況に応じてキャラクタの挙動をコントロールする処理のことを AI と呼ぶ．

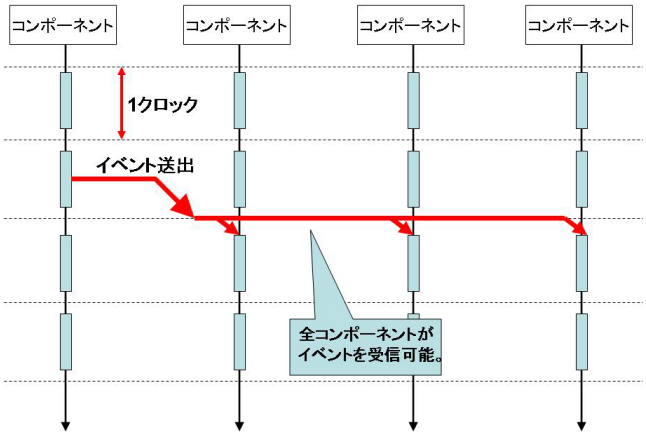

図 2.29: Reactive Programming のコンポーネントとイベントの処理の流れ

データバインディングは決められた要素同士を 1 つの関係で結びつけるので，上記の例のように機能が単純な要素同士の通信には適している．しかし，キャラクタの種類や状態に応じて変化する，キャラクタ間の通信の記述には十分であるとは言えない．

2.6.6 Reactive Programming

Reactive Programming[Sus06] は，プログラム要素の変化に応答する処理の記述を容易にするためのプログラミングモデルであり，Java の Sugar Cube[BS98] 等，いくつかのプログラミング言語にライブラリとして実装されている．

Reactive Programming ではコンポーネントと呼ばれる処理単位の間で，イベントと呼ばれる通知により通信が行われる．コンポーネントはクロックと呼ばれる Reactive Programming 上の論理的な短い時間単位で少しづつ自分の処理を進める．あるコンポーネントがあるクロック時に変化しその変化をイベントとして送出すると，イベントは次のクロックで他のすべてのコンポーネントに通知される．各コンポーネントはクロックで処理が再開される度に通知されたイベントを処理する (**図 2.29**)．

Reactive Programming はデータバインディングのデータ同期モデルをデータだけでなくプロセスを含むコンポーネントに拡張したものである．各コンポーネントの処理は，クロックにより細分化されており，キャラクタの処理をアニメーションフレーム毎に少しづつ進ませるゲームプログラムと類似している．また，データバインディングと違いコンポーネント同士は直接関係付かず，イベントは全てのコンポーネントに送信され，あるイベントが処理

されるかどうかは各コンポーネントが実行時に検査する．この仕組みにより，1 つのイベントを複数のコンポーネントが処理することも可能である．

しかし，Reactive Programming は，イベントが全コンポーネントに通知される仕組みであるため，コンポーネントの種類や状態に応じてイベントを通知することはできない．また，コンポーネントの実行順序やイベントが送信順序は定義されておらず，通信の処理順序を明示することもできない．

2.6.7　ルールベースシステム

ルールベースシステムは，与えられる情報について，予め準備したルール群を適用して適切な処理を選択する計算モデルで，エキスパートシステム [鷹田 97] などで用いられている．Jess[FH03] のようにライブラリとしての実装もある．

ルールベースシステムは，与えられる情報を，条件判定とその処理を組にした複数のルールに与え，各ルールは条件を満たせば処理を実行する．複数のルールが同時に満たされる場合のために衝突解決の仕組みも用意されており，ルールの処理順序が明示できる．

ルールベースシステムのルールをゲームルールと見ると，ルールの条件は，キャラクタの状態や衝突の判定等，ルールの処理部分は，条件成立時にキャラクタについて行う処理と見ることができる．よって，キャラクタの種類や状態に応じて変化する通信の記述に適していると考えられる．

しかし，ルールベースシステムは並行処理間の通信のためのモデルではなく，並行処理間でいつどのようにルールを適用するべきかについては考慮されていない．また，個々の並行処理が直接他の処理へ通信を発するような記述は難しい．

2.6.8　Tuple Space

Tuple Space は分散システムのための通信モデルで，プログラミング言語 LINDA[Gel85] で最初に提案された．多くの並列／分散処理が Tuple Space を用いることで簡潔に記述でき，Tuple Space の機能はプログラミング言語 LINDA とは独立していることから，LINDA 以外の言語にも採用され [Cia94, DRW95]，機能拡張に関する研究も複数なされている [Gel89, RW98]．

Tuple Space はすべての処理から共通にアクセスできる場所で，この中に，名前と複数の値を組にした Tuple と呼ばれる構造化されたメッセージを読み書きすることで通信を行う (**図 2.30**)．

LINDA では，Tuple Space の読み書きに使用するのは次の 5 つの命令である．

out 命令　任意の引数を組にした Tuple を Tuple Space へ書き込む．`out` を実行した回数だけ Tuple は書き込まれる．

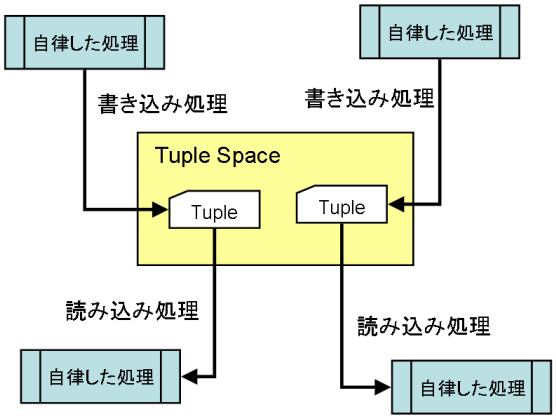

図 2.30: Tuple Space の概念図

in 命令 Tuple Space から指定した引数にマッチする Tuple を読む．このとき読んだ Tuple は Tuple Space から消去される．マッチする Tuple が無いと，プロセスは待たされる．

rd 命令 `in` と同じく Tuple の読むが，`in` と異なり Tuple は Tuple Space に残される．マッチする Tuple が無いと，プロセスは待たされる．

inp 命令 `in` と同じだが，マッチする Tuple がない場合はエラーを返しプロセスは停止しない．

rdp 命令 `rd` と同じだが，マッチする Tuple がない場合はエラーを返しプロセスは停止しない．

Tuple Space には次の特徴がある．

- 通信を簡潔に記述できる
- シンプルで理解しやすい

汎用的プログラミング言語で通信を記述する場合，変数やメソッド呼び出し等のいくつかの言語機能から，通信の内容に応じて，選択的に組み合わせて使用しなければならず，多数の処理への通信や，時間的に分離した通信の記述は，わかりにくく複雑なものになりやすい．

一方，Tuple Space を用いる場合，通信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．受信のためのメソッド等を必要とせず，Tuple の読み書きのみで通信を記述するので，通信により制御の流れが多数のプログラムに渡ることがなく，個々のプログラムの簡潔さを維持しやすい．

さらに Tuple Space は，全ての並行処理間の通信を Tuple の読み書きだけで表現するので，シンプルで理解しやすい．よって，プログラミングに精通していなくても理解することができ，ゲームデザイナーが利用するのにも適している．

しかし，Tuple Space では，Tuple の処理順序は並行処理の処理順序に依存し，並行処理の処理順も定義されておらず，通信の処理順を明示することは難しい．また，Tuple Space は汎用的で Tuple の読み書きというシンプルなモデルであるため，複雑な通信を実装する際は多くの Tuple のやり取りが必要になり，プログラムも読み込みや書き込み処理が多く，間違いが起きる原因になりうる．

2.6.9 Join Calculus

join calculus[FGL+96, IR] は，分散システムのために提案された並行計算モデルである．複数の並列動作するプロセスが，メッセージの送受信を行うチャネルを通して通信しながら計算が進行する．また，join pattern と呼ばれる，複数の異なるメッセージを同時に受信する記法により，プロセスの処理と通信を分けて記述でき，プロセスを配線で繋いでいくようなプログラミングが可能である．

次は，プログラミング言語 OCaml に join calculus を統合した JoCaml[FFMS08] によるプログラムの例である．この例では，引数として渡された値を表示する echo という名前のチャネルと，echo に値を渡すプロセスの起動を行う．

```
def echo(x) = print_int x; 0 ;;  (* チャネル echo の定義 *)

spawn echo(5);;            (* echo に 5 を渡すプロセスを起動 *)
spawn echo(3) & echo(9);; (* 二つのプロセスを同時に起動 *)
```

上記の例のチャネルは，他のプログラミング言語の手続きに似ているが，メッセージを送ったプロセスと並行した新しいプロセスとして起動される点が異なる．そのため，メッセージを送ったプロセスはチャネルの処理とは無関係に処理を継続できる．

JoCaml では，次のように join pattern と呼ばれる記法を使って，1 つのチャネルが複数の異なるメッセージを同時に受信する処理を記述ができる．

```
(* bomb と ignite のメッセージを受けると処理を始めるチャネル *)
def bomb(n) & ignite() =
  print_int n;
  print_endline " bombs exploded!"; 0;;

(* 2 つのプロセスを起動 *)
spawn bomb(3);;   (* bomb を送る．何も実行されない *)
spawn ignite();;  (* "3 bombs exploded!"と表示される *)
```

チャネルは値を返すこともできる．次の例は，2 つのプロセスから適当な整数値をメッセージとして受信し，その合計値を元の 2 つとは異なるプロセ

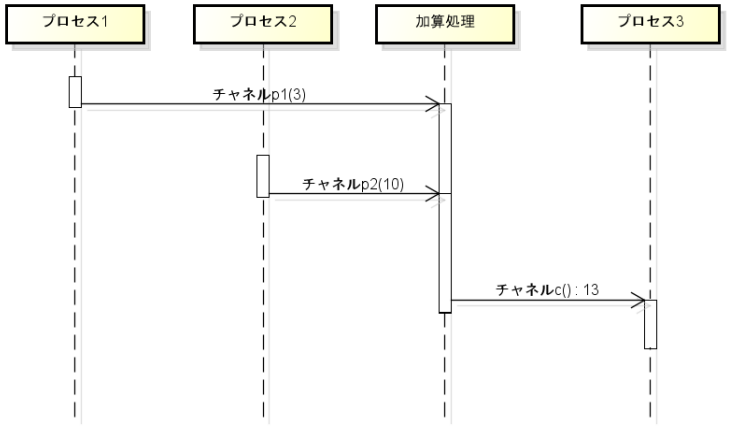

図 2.31: JoCaml プログラム例の 3 つのプロセスのシーケンス図

スへ返す例である．チャネルから値を得ようとすると，プロセスは値が返ってくるまで停止する．

```
(* p1 と p2 を通して 2 つの整数を取得し,c を通して合計を返す *)
def p1(m) & p2(n) & c() = reply (m+n) to c;;

(* 整数を渡すプロセスを 2 つ起動 *)
spawn p1(3);;
spawn p2(10);;

(* 合計値を得て表示するプロセス *)
spawn
  let sum = c() in   (* チャネル c で合計値をもらう *)
  print_int sum; 0;; (* 結果表示 *)
```

このプログラムの動作を**図 2.31** に示す．3 つのプロセスが協調して動作するプログラムを，JoCaml ではシンプルに記述できている．

JoCaml にはこの他にも OCaml に由来する，高度なチャネルのパターンマッチング機能やデータを構造化する機能があり，これらを用いて複雑なプロセス間通信を，他のプログラミング言語に比較して平易に記述可能である．

join calculus の，複数のメッセージを同時に受信してメッセージに応じた処理をするモデルは，複数の自律した処理を持つキャラクタ間の通信を表現する方法として期待できる．

しかし，join calculus の，プロセスがメッセージをチャネルを通してやりとりし，チャネルはプロセスとして起動するというモデルは，複雑で理解や制御が難しく，プログラミングに精通していないゲームデザイナーにも理解が容易な通信モデルとして，適切とは言えない．また，join calculus のプロセスは処理順序が定義されていないので，チャネルの処理順序を明示することはできない．

2.6.10　ゲーム記述に適した通信モデルのまとめ

本節では，ゲームキャラクタ間の通信を表現するのに必要な性質を挙げ，計算機科学分野で研究されている機構や通信モデルについて検討した．**表 2.5** に説明した通信モデルについてまとめた．

手続き呼び出しや RPC は，多くのプログラミング言語で提供される機能であることから，ゲームプログラミングやゲームシステム記述ツールでもキャラクタ間の通信を記述する方法としても利用されている．しかし，排他的で階層的な処理構造は自律した処理同士の通信には適していない．

コルーチンも，ゲームプログラミングで AI 等の実装に有用な方法として知られているが，手続き呼び出しと同様，呼び出し側と呼び出される側という関係があり，並行処理とその間の通信の記述に適切ではない．

イベント駆動型プログラミングは，特定のイベントに対応する処理を記述するのに適しており，ゲームプログラミングやゲームシステム記述ツールで広く使用されている．しかし，システムが用意しない固有のイベントの記述が難しく，キャラクタの種別や状態に依存した通信の記述に適しておらず，イベントの発生タイミングや処理順序を明示することはできない．

データバインディングは，決められた要素間の通信に限られ，機能も単純なことから，キャラクタ間の通信記述には適さない．

Reactive Programming は処理モデルはゲームプログラムと類似しているが，イベントの通知対象が常に全コンポーネントであり，種類や状態に依存した記述が容易ではなく，処理順序の明示もできないため，キャラクタ間の通信に適したモデルとは言えない．

ルールベースシステムは，並行処理のためのモデルではないため，キャラクタ間の通信モデルとして適用することは難しい．

Tuple Space は様々な通信を簡潔なモデルで表現できるが，処理順序が明示できないことと，汎用的でシンプルなために，複雑な通信には煩雑な Tuple の操作が予想される．

join calculus もプロセス間通信を平易に記述可能であるが，通信毎にプロセスが生成されることから多くのプロセスを管理しなければならず，わかりやすさの点で問題がある．

手続き呼び出し，コルーチン，イベント駆動型プログラミングは，現在のゲーム開発においてゲームシステム記述に利用されている．しかし，どれもキャラクタ間の通信モデルとしては不十分で，ゲームシステムが大きく複雑になると，記述も見通しが悪く，可読性の悪いものとなる．また，本節では述べなかったが，2.4 節のアクションゲーム A やシューティングゲーム S1 のように，グローバル変数をキャラクタ間の通信に使用することもできる．しかし，グローバル変数は情報を全てのキャラクタで共有する手段としては有効でも，通信元や通信先，通信の処理を記述することはできないので，キャラクタ間の通信の記述には適していない．

表 2.5: 通信モデルのまとめ

名前	理解のしやすさ	複数の並行処理間の通信	種類や状態に依存した処理	処理順序の明示
手続き呼び出し，RPC	○	×	×	○
コルーチン	△	×	×	×
イベント駆動型プログラミング	○	×	×	×
データバインディング	○	×	×	×
Reactive Programming	○	○	×	×
ルールベースシステム	○	×	○	○
Tuple Space	○	○	△	×
join calculus	×	○	△	×

2.7 まとめ

本章では，既存のプログラミング言語や開発ツールについて，その詳細と，1 章で述べた 4 つの機能的な要件を満たしているかどうかについて説明した．

2.2 節では，ビデオゲーム開発を目的としたプログラミング言語について述べ，商用のゲームアプリケーション開発に利用されているにも関わらず，どの言語も 1.9 節の 4 つの要件を十分に満たしていないことを説明した．

2.3 節では，ビデオゲーム開発を目的とした統合開発環境について述べた．統合開発環境は，高度なプログラム開発の必要なしにゲームアプリケーションの開発が可能なので，開発コストを大幅に低減することができる．しかし，これらのツールが記述用に提供しているプログラミング言語は，1.9 節の 4 つの要件を満たしておらず，ゲームデザイナーがトライアンドエラーを繰り返しながら開発をするのに十分ではないことを説明した．

2.4 節では，個々のゲーム開発専用に作られた記述ツールについて説明した．これらの記述ツールも 1.9 節の 4 つの要件を満たしておらず，また，要件を満たす機能についても，目的とするゲームに特化しすぎており，他の種類のゲームシステム記述には適していないものがあることを述べた．

2.5 節では，ビデオゲームとは異なる他分野のアプリケーション開発ツールについて説明した．これらのツールは，目的がプログラミングの教育や，アニメーションの作成等であることから，他のツールに比較して，わかりやすくアプリケーションの制作が容易である一方，1.9 節の 4 つの要件を十分に満たしていないことから，ゲームシステムの記述には適していないことを述べた．

特に，1.9 節の 4 つの要件のうち，キャラクタ間の通信を記述するのに十分な機能を持つツールは無い．そこで，2.6 節において，キャラクタ間の通信を表現するのに適切な通信モデルの性質について議論した．また，ビデオゲームにおいてゲームキャラクタが自律して動作しながら通信を行うという処理モデルが並行処理プログラミングの計算モデルと合致することから，現在，計算機科学分野において研究されている並行/並列処理間の通信モデルについて調査し，その特徴やゲームシステム記述への適用可能性，現在ゲーム開発に利用されている通信モデルについて議論した．

これらの議論から，ビデオゲーム開発において，現在のゲームシステム記述ツールは，ゲームデザイナーがトライアンドエラーを繰り返しながらゲームを開発するために必要なツールであるにも関わらず，ビデオゲームシステムの記述に必要な要件を十分に満たしていないという結論を得た．

記述機能が十分でなければ，目的とするゲームの記述が難しくなり，トライアンドエラーの障害となりうる．アイディアを実装して動かすまでにも時間がかかり，開発イメージを共有する目的にも利用しづらい．

そこで，本章の議論を元に，1 章で述べた要件を満たすべく，新しいプログラミング言語の設計と開発を行い，ビデオゲーム開発に適用し評価を行った．次章では，この新しいプログラミング言語とその評価について説明する．

第3章　ゲームシステム記述に適したプログラミング言語の開発

3.1　はじめに

1 章では，トライアンドエラーを繰り返すビデオゲーム開発にはゲームシステム記述ツールが不可欠であり，十分な記述性のためにはゲームシステム記述に適したプログラミング言語が必要であることを述べた．また，ゲームシステム記述用のプログラミング言語には,「キャラクタの状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」の 4 つの記述機能が求められることについても述べ，2 章において，既存の言語ではこの 4 つの要件が十分に満たされていないことを説明した．

4 つの要件を満たすプログラミング言語を開発する方法として，要件を満たすよう，既存のプログラミング言語に修正をする方法も考えられる．しかし，既存のプログラミング言語は 4 つの要件を十分に達成していないことから，記法や構文の大幅な修正が必要となる．また，修正の結果，言語仕様が複雑で理解の難しいものとなり，プログラミングに精通していないゲームデザイナーによる利用を難しくすることも考えられる．

そこで，4 つの要件を満たすように新しいプログラミング言語を設計し開発することにした．全く新しいプログラミング言語とすることで，各要件の記述方法を，既存言語の影響を受けない適切で簡潔な構文として設計することができる．また，適切で簡潔な構文は，言語の理解を容易にし，ゲームデザイナーにも利用しやすいものとなることが期待できる．

本章では，この新しいプログラミング言語 S540 と，S540 を実装した S540 記述ツール [西森 03] について説明する．以下，3.2 節で S540 の設計について，3.3 節で S540 の詳細について説明する．3.4 節では，S540 の組み込み方法と実装について説明し，3.5 節で S540 を実際のゲーム開発プロジェクトへ適用したことと，その結果に基づいた評価について述べる．3.6 節では，S540 を使用した開発プロジェクトと，他の S540 を用いていない開発プロジェクトの比較を行い，3.7 節で S540 の評価について述べる．最後の 3.8 節で本章の総括をする．

S540 の構文定義は，本論文の末尾 A.1 節に付録として掲載している．また，S540 を学習するために用意した入門用テキストを A.2 節に掲載している．

3.2　S540 の設計

3.2.1　設計方針

S540 は前述のように，ゲームデザイナーが容易にゲーム記述を行えるようにすることを目的としたプログラミング言語である．S540 とその記述ツールの設計に当たっては次の方針を採用した．

- ゲーム記述に必要な 4 要件である「キャラクタの状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」の記述がしやすい機能を持つ．
- プログラミングの経験が無くとも利用しやすい設計とする．
- 実行時の処理速度を考慮した実用性の高いものを目指す．

S540 は，1.9 節で述べた 4 つの要件「キャラクタの状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」について，明示的で簡潔な構文を持つ設計とし，ゲームデザイナーにとって直感的でわかりやすいプログラミング言語となるようにした．

プログラミングの経験が無くとも S540 を利用しやすいよう，記述方法には日本語文字を取り入れて，普段開発で使用している用語を記述に直接取り入れられるようにしている．S540 の明示的で簡潔な構文は，記述の容易さと同時に利用のしやすさも目的としている．

ビデオゲームアプリケーションは，1/10〜1/60 秒程度の短い時間で全キャラクタについてゲームシステムや 3D グラフィクス等の処理をしなければならないので，可能な限り処理の高速性がが求められる．そこで，S540 記述ツールでは，記述されたゲームシステムプログラムを C 言語プログラムへ変換することで，実行時に高速に動作するようにした．

次節からは，S540 における 1.9 節で述べた 4 つの要件の扱いと，S540 記述ツールの設計方針について説明する．

3.2.2　キャラクタの状態遷移処理

ゲーム中のキャラクタは,「歩いている状態」「ジャンプ中の状態」等多くの「状態」となりうる．この状態の処理は，状態がゲームキャラクタの挙動の記述単位であり，キャラクタそれぞれが固有に持つことから，キャラクタの定義内で明示的で簡潔に記述できることが望ましい．

そこで，S540 ではキャラクタごとの状態遷移を次のような形式で記述できるように設計する．

```
キャラクタの定義
  状態 1
        状態 1 の処理
            :
        状態 2 への遷移
            :
  状態 2
        状態 2 の処理
            :
        状態 1 への遷移
            :
キャラクタの定義終わり
```

各状態の処理内では，他の状態へ遷移する命令を記述できるようにする．状態遷移処理を記述するのに適した機能のないプログラミング言語では，1.9 節の**図 1.4** のように，状態を表す変数や条件分岐を使用して記述しなければならず，状態の定義の見通しが悪くなり，間違いの原因となりやすい．上記の方法は，状態を表す変数や，各状態を処理するための分岐処理もなく，状態遷移処理を明示的にわかりやすく記述できる．

3.2.3 時間に沿った処理

ゲーム中のキャラクタは,「段々大きくなる」「10 メートルを 3 秒で移動して消える」「構えて 1 秒後に攻撃」等のアニメーション処理をする．これらの処理の記述のために，時間に沿った記述が容易な機能が必要である．

アニメーションオーサリングツールである Alice[CAB+00] や，プログラミン [文部 10] には一定時間処理を停止する命令があり，アニメーションの開始や終了のタイミングを記述しやすい．そこで S540 では，次の「待つ」のような記述方法で，指定時間実行を一時停止したり，条件を待ったりする記述を導入する．

```
ある状態の処理
    :
    歩く
    待つ 60
    走る
    待つ 行き止まり検査
    歩く
    :
```

この例は「歩き出して 60 フレーム経過したら走り，行き止まりになったら歩く」という時間に沿った処理を記述している．1 つ目の「待つ 10」の部分は処理を 10 フレーム停止させる記述で，2 つ目の「待つ 行き止まり検査」は「行き止まり検査」という条件式が満たされるまで処理を中断する記述であ

る．このように「待つ」の導入により，時間に沿った処理をわかりやすく記述できる．

また，高度なリアルタイムゲームの場合，アニメーションのコマに相当するフレーム単位で細かく処理を実装することもあるため，時間の単位としてはフレーム数が適当である．そこで，待つに指定できる時間には，フレーム数を指定するようにする．

この機能が無いプログラミング言語を使って時間に沿った処理を記述すると，1.9 節の**図 1.6** のように，経過時間を管理する変数と条件分岐を用いた，可読性の悪いものになる．上記のように上から下への処理の流れを時間の流れと対応させられれば，記述が容易で，わかりやすく処理内容を把握しやすいプログラムとなる．

3.2.4　並行処理

ゲーム中のキャラクタは自律的に動作するので，S540 ではキャラクタを出現させると，自動的に定義された状態遷移処理を他の処理とは並行に開始するモデルとする．

さらに，1 つのキャラクタについても,「走りながらジャンプ」「ジャンプしながら攻撃」等，複数の処理を同時に行うことがある．これは,「走る⇔歩く⇔停止」「地面⇔ジャンプ (空中)」「通常⇔攻撃⇔防御」のような状態遷移が,複数並行して組み合わさってキャラクタの状態となっているものと定式化できる．

S540 ではこのような記述も容易となるよう，1 つのキャラクタに複数の状態遷移を記述できるようにする．次は 2 つの並行する処理を「あるキャラクタ」というキャラクタ定義内で記述する例である．

```
あるキャラクタ
    :
  状態遷移処理の記述
    :
==================
    :
  状態遷移処理の記述
    :
キャラクタの定義終わり
```

3 つ以上の任意の数の=を並べた記号 “==========” でキャラクタ内部を区切ることで，区切られた上下の部分を並行する異なる処理であることを示す．このキャラクタは出現すると，定義された 2 つの状態遷移処理を並行して実行する．

適切な機能を持たない言語で並行処理を記述すると，1.9 節の**図 1.9** のように，並行処理が明示できず，個別の並行処理を呼び出すための手続き等を必要とするような，可読性の悪いものになる．上記の方法では，並行処理を

管理するための変数や呼び出し記述も必要ない．さらに，並行処理をこのような区切り記号で平易に記述できる機能は，他のプロセスやスレッド等の並行処理を利用できるプログラミング言語にも無く，ゲームシステムプログラムの簡潔さの維持に効果があると期待できる．

3.2.5 キャラクタ間の通信

キャラクタ間の通信には，2.6.1 節で述べたように,「攻撃が当たったら，敵にダメージを与える」のような 1 対 1 のものもあれば,「プレーヤーに向かって複数の敵キャラクタが組織立って攻撃」等の 1 対多，多対多の通信がある．「ボスが死んだら子供も死ぬ」場合，ボスの子供がプレーヤーにより破壊されうるなら，通信対象となるキャラクタ数は不定であり，存在しないこともある．

ゲーム中の通信やキャラクタ間の関係処理にはその他にも様々なものがある．ケースに応じた機能を導入することも考えられるが，言語の要素が増えることは学習や理解の容易さの点で問題があり，プログラミングに精通していないゲームデザイナーのためのツールとしては不適切である．

そこで，様々な通信をシンプルな読み書き操作のみで表現可能で，受信メソッド等を必要とせず，通信プログラムのために制御の流れが複雑になるのを避けることができ，さらにシンプルなモデルであることから，プログラマーでなくとも理解が容易であると考えられる，Tuple Space をキャラクタ間の通信モデルとして採用する．

3.2.6 その他の機能

前述の 4 つの機能的要件以外に，S540 と S540 記述ツールは次のような点を配慮して設計を行う．

- 日本語文字で記述可能にする．
- ツールによる記述部分がアプリケーションの上位となる設計とする．
- 特定のゲームに特化しない実装とする．
- 実行速度のために，ゲームシステムプログラムを C 言語プログラムへ変換するトランスレーターを用意する．

新しい言語のユーザーであるゲームデザイナーはプログラマーではなく，他のプログラミング言語の記述に使用される英単語には親しみがない．また，ゲームデザインや開発を進める上で使用する言葉を直接 S540 の記述に利用できれば，デザインとプログラミング言語による記述の間に直接的な対応付けがしやすくなり，開発はより円滑になると考えられる．そこで，言語は日本語文字を使った記述が可能にする．

ゲームシステム全体を記述可能とするためには，S540 で記述したプログラムが，アプリケーションの上位層になければならない．そこで，S540 プログラム実行系がアプリケーション下位層のプログラムを呼び出し，ゲームシステムプログラムはこの実行系の上で動作する構造とした．**図 3.1** は，この構造を図示したものである．

また，ビデオゲームのハードウェアやソフトウェアの技術は常に進歩しているため，グラフィクスやハードウェアのコントロールを行うアプリケーション下位層は個々の製品により異なる．そこで，これらの機能は S540 では提供せず，代わりに S540 プログラム実行系をアプリケーション下位層から分離された実装とすることで，様々な製品開発で S540 記述ツールが利用しやすい構成にする．

実行時の処理時間のコストを最小限にするために，S540 で記述されたゲームシステムプログラムを直接 C 言語プログラムへ変換し，ゲームシステムプログラムが高速に実行されるようにする．

次節では，S540 の詳細について述べる．

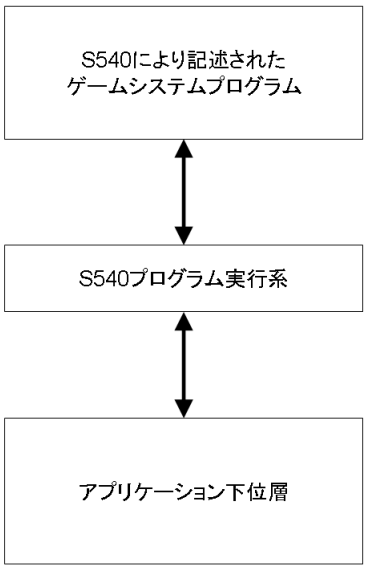

図 3.1: S540 によるアプリケーション構造図

3.3　S540 の詳細

3.3.1　S540 の構造

ゲームはゲーム中のキャラクタが決められたゲームシステム上の仕組みやルールに沿って，自律的に動作しながら進む．そこで，S540 は，ゲームのキャラクタを基本的な単位として記述する構造にした．

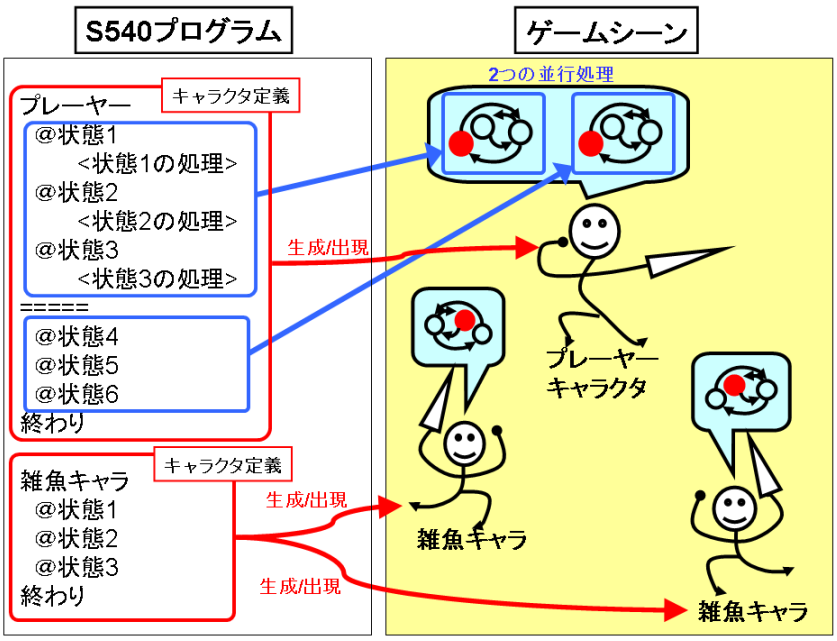

図 3.2: S540 プログラムの構造

図 3.2 は S540 で記述するプログラムの構造と，プログラムとゲームシーンの対応を図にしたものである．

プログラムはキャラクタの定義のみで構成される．**図 3.2** には，2 種類のキャラクタ「プレーヤー」と「雑魚キャラ」を定義した例である．キャラクタ定義を使って，キャラクタを生成し，ゲームシーン中に出現させることができる．

各キャラクタの定義には，1 つ以上の並行する処理の記述ができる．各並行処理は，キャラクタが生成されゲームシーン中に出現すると自動的に処理を開始し，自律的に処理を進める．

1 つの並行処理は 1 つ以上の状態を含む状態遷移処理として記述される．2 つの並行処理が記述されたキャラクタは，2 つの状態遷移処理を持つことになる．

各状態内部には，キャラクタがその状態で実行する命令や文を記述する．記述には，時間に沿った処理や制御構文を使った条件分岐や繰り返し処理を使用することができる．また，異なる状態への遷移も状態内部で記述する．

次節からは S540 の詳細について説明する．記述に使用する文字はプログラムの最も基本的な要素であるので，次の 3.3.2 節では，日本語文字による記述機能について説明する．次の 3.3.3 節で，1.9 節の要件の 1 つであるキャラクタとその状態遷移処理に対応する記述機能について説明する．状態内部ではいくつかの制御構文を使用することができるので，続く 3.3.4 節では制御構文についてまとめて説明する．3.3.5 節，3.3.6 節，3.3.7 節では，1.9 節

表 3.1: 同一とみなされる予約語の一覧

作る	create	同期	sync
消す	erase	組み合わせ	each
取る	in	プリント	print
読む	rd	整数	int
書く	out	実数	float
待つ	wait	文字列	string
取ってみる	inp	論理	bool
読んでみる	rdp	キャラクタ	character
条件ループ	while	自分	self
無限ループ	infloop	且つ	and, かつ
ループ	times	または	or
終わり	end, おわり, オワリ, 終わり, 終り	真, 正しい	true
ループ脱出	break	偽, 間違い	false

の要件の残る 3 つの要件「時間に沿った処理」「並行処理」「キャラクタ間の通信」に対応する記述機能について説明する．3.3.8 節では，簡単なプログラム例に基づいて S540 による具体的なゲームキャラクタの記述方法について述べ，最後の 3.3.9 節で，S540 を用いて記述がどのように改善されるかについて，汎用プログラミング言語である Java の例と比較して検討を行う．

3.3.2　日本語文字による記述機能

S540 は，ゲームデザイナーが開発で使用している言葉をそのまま S540 を使った記述でも用いることができるよう，プログラムに ASCII 文字だけでなく日本語文字を用いた記述を可能にした．また，言語のほとんどの予約語についても，ASCII 文字と日本語文字の両方を用意した．この対応を**表 3.1** に挙げる．

また,「＋」や「＊」は半角文字としても全角文字としても存在する．これらは記述者には同じ文字として見えるため，記述ツールが違う文字として区別すると，記述間違いの発見を難しくする [中谷 02]．日本語入力システムの設定状況によっては，ピリオドを入力するよりも「。(句点)」を入力するほうのが簡便な場合もある．さらに，日本語入力システムを用いている場合，アルファベットの大文字と小文字を区別して入力するには，煩雑な手順を踏まなければならないことがある．

そこで，半角文字と全角文字の両方ある文字と，同じ文字でなくとも，入力上の利便性の点で同一文字として認識するほうがよい文字は同一の文字と認識することにし，アルファベットに関しても，大文字と小文字の違いを区別しないことにした．特に，異なる文字でも同一文字としているものを**表 3.2** に挙げる．

表 3.2: 同一視される文字の一覧

.(ピリオド)	。(句点)
,(カンマ)	、(読点)
*(アスタリスク)	×
/(スラッシュ)	÷
「」	[] 『』

3.3.3 キャラクタと状態遷移処理の記述機能

S540 のキャラクタの定義は他の汎用的なオブジェクト指向言語のクラス定義に近いが，クラス定義中では，2 章で述べた Unreal Script の State 機能のように，キャラクタの個々の状態を 1 つのルーチンとして明示的に記述する．

本節では S540 での，キャラクタとキャラクタの状態遷移処理の記述方法について述べる．

キャラクタと状態の定義

キャラクタは記述者が適当に決めるキャラクタ名で始まり「終わり」で終わる．キャラクタ内部に「@名前」で始まる状態の記述を列挙する．次は，キャラクタの簡単な記述例である．

```
あるキャラクタ
  @立ち状態
    # 立ち状態の処理
  @歩き状態
    # 歩き状態の処理
終わり
```

この例は,「あるキャラクタ」という名前のついたキャラクタの記述で，このキャラクタは「立ち状態」と「歩き状態」という 2 つの状態を持つ．@名前から次の@記号の直前までが，状態の処理を記述する部分になる．

#記号はプログラムを読みやすくするために利用できる注釈の開始記号で，S540 処理系はこの記号から行末までを無視して解釈を行う．

処理の実行順序

状態内部の処理は，処理の流れがわかりやすいよう，上から順に実行されるようにした．

また，キャラクタ内部で一番最初に記述されている状態は，キャラクタがゲームシーンに出現したときの最初の状態とした．これにより，キャラクタ

の最初の実行箇所を指定する命令等を必要としない，簡潔な記述ができるようになっている．最初に実行される状態を「初期状態」と呼ぶ．

次は，数字の１から５までを順に表示する「１から５まで表示」という名前のキャラクタの記述例である．

```
１から５まで表示
  @最初の状態
    プリント "1"
    プリント "2"
    プリント "3"
    プリント "4"
    プリント "5"
  @違う状態
    # その他の処理
終わり
```

「１から５まで表示」キャラクタには２つの状態「@最初の状態」と「@違う状態」が記述されている．S540 では最初に記述されている状態が初期状態なので，このキャラクタは生成後「@最初の状態」から実行を開始する．

状態の範囲は,「@名前」から，次の「@名前」かキャラクタの記述の最後を表す「終わり」までである.「@」は他の定義に使用できない記号としており，状態の区別は@記号だけで区別することができ,「{}」等を使用して範囲を指定する必要はない．

処理が状態の最後まで到達した場合，処理はそこで停止し再度実行されることはない．これにより，制御の流れが予期しないプログラムへ移るバグの発生を防ぐことができる．

プリント文はコンソールに指定された文字列を表示する命令である．プリント文は，手軽にプログラムの動作状態を確認するのに便利であることから S540 の機能として用意した．

状態遷移の記述

状態間の遷移を記述するために GOTO 文を用意した．次は，GOTO を使用した，状態間の遷移をする記述の例である．

```
あるキャラクタ
  @立ち状態
    # 立ち状態の処理
    GOTO @歩き状態
  @歩き状態
    # 歩き状態の処理
    GOTO @立ち状態
終わり
```

上記の例では「@立ち状態」の処理の実行後「@歩き状態」へ遷移する．同様に「@歩き状態」の処理が終わると「@立ち状態」へ遷移する．

キャラクタ固有のデータの定義

ゲームキャラクタは体力や攻撃力等の固有のデータを持つ．S540 にはこのような各キャラクタの情報を保持するために，多くのオブジェクト指向言語と同様に，キャラクタ固有の変数を定義する機能を用意した．

次は，キャラクタ固有のデータとして「体力」と「攻撃力」を「あるキャラクタ」というキャラクタに定義する例である．

```
あるキャラクタ
  体力 : 整数
  攻撃力 : 整数
  @初期化処理
    体力 = 100
    攻撃力 = 50
     GOTO @立ち状態
  @立ち状態
    # 立ち状態の処理
終わり
```

データは，最初の状態の記述の直前に「データ名：データの型」という形式で記述する．データ型には整数のほかに実数，文字列，ベクタ[1]，論理[2]，キャラクタ型などが使用できる．これらのデータは各処理中で参照や代入が可能となっている．

また，状態内部で次の記述をすることで，状態内部でのみ使用可能なデータの定義もできる．

```
あるキャラクタ
  @ある状態
    カウント : 整数
    カウント = １０
    プリント　カウント
終わり
```

ここで定義している「カウント」は「@ある状態」内部でしか使えないデータである．

キャラクタをゲームシーンに出現させる記述方法

「あるキャラクタ」という名前で定義されるキャラクタを，ゲームシーン中に出現させるには，次の記述をする．

[1] キャラクタの位置などを表すために実数 3 つが組となったもの
[2] 真偽の 2 値のみで表現されるデータ型

```
作る あるキャラクタ
```

「作る」命令が実行されると，指定した名前のキャラクタが「生成」され，ゲームシーン中に出現する．1 つの定義について何度も「作る」命令を実行することで，同一キャラクタを複数個，ゲームシーンに出現させることができる．

プログラム起動時に作られるキャラクタの記述方法

「ROOT」という名前のキャラクタに対しては，プログラム起動直後に自動的に 1 度だけ「作る」が実行される．これによりプログラム起動時の処理を管理することができる.「ROOT」キャラクタはこの特殊な機能以外は他のキャラクタと違いがない.

3.3.4　制御構造の記述機能

「ボタンを押したら攻撃状態に遷移」等の条件付きの状態遷移処理や，同じ動きを繰り返す処理の記述のために，S540 には，他の多くのプログラミング言語にある，制御構造を記述する次の文を用意した.

- 条件によって処理を変えるための IF 文.
- 繰り返し処理のための，条件ループ文，ループ文，無限ループ文.
- 複数キャラクタやその組み合わせについての処理をする組み合わせ文.
- 繰り返し処理や組み合わせ文の処理を終了するループ脱出文.

これらの文は，キャラクタの状態内部の処理として記述する．キャラクタや状態の定義の外側で使用することはできない.

IF 文

様々な条件によって処理を変えるための記述方法として IF 文を用意した.次の例は,「体力」というデータが 0 以下の場合だけ「@死亡状態」へ遷移する記述である.

```
IF 体力 <= 0
  GOTO @死亡状態
END
```

次の「ELIF」や「ELSE」を使うと条件を列記することができる.

```
IF 体力 <= 0
  GOTO @死亡状態
ELIF 体力 <= 10
  GOTO @弱り状態
ELIF 体力 <= 50
  GOTO @やや弱り状態
ELSE
  GOTO @普通状態
END
```

条件は常に上から順に評価され，一番最初に成立した条件の処理が実行される．また，最後に「ELSE」を使うことですべての条件が成立しない場合の処理を記述できる．

条件ループ文

ゲーム中には,「プレーヤーキャラクタがいる限り，攻撃を繰り返す」等，ある条件下で同じこと繰り返す処理が必要となることが多いので，S540 には条件を指定して処理を繰り返す条件ループ文を導入した．

次の例は 1 から 10 までの和を求めるプログラムである．

```
和 = 0
カウント = 1
条件ループ カウント <= 10
  和 = 和 + カウント
終わり
プリント 和
```

ループ文

ある回数，同じことを繰り返す処理は,「3 回攻撃して，一旦逃げる」のようにゲーム中でも多く必要となることが予想される．そこで，S540 には指定回数処理を繰り返すための記述方法として「ループ」文を導入した．ループ文は条件ループ文と変数を併用することでも記述可能だが，可読性と記述性が良くなると期待できる．

指定回数処理を繰り返すための記述方法として「ループ」文がある．次の例は 2 の 10 乗を求めるプログラムである．

```
積 = 1
ループ 10
  積 = 積 * 2
終わり
プリント 積
```

無限ループ文

「ずっとプレーヤーを追いかけ続ける」「攻撃後，一旦逃げて，また攻撃，を死ぬまで繰り返す」のように，ある動作を無限に続ける動作はゲーム中で多く必要とされる．そこで，S540 には無限に繰り返す処理を記述する「無限ループ」文を導入した．

無限ループ文もループ文と同様，条件ループ文で置き換えることができるが，ある処理を無限に繰り返すことを明示的に記述可能にすることで，可読性の向上を狙っている．次は,「処理」と表示し続ける無限ループの記述例である．

```
無限ループ
  プリント　"処理"
終わり
```

無限ループ文は，3.3.5 節で述べる「待つ」や「同期」命令を利用することで，一定時間おきの処理や毎フレーム繰り返す処理を記述できる．

複数キャラクタやその組み合わせについての処理

ある種類のキャラクタや，複数のキャラクタに対して一括した処理を行うために,「組み合わせ」というループ文が用意した．次の例はゲーム中に存在するすべての「ロボット」キャラクタの情報をコンソールに表示する．

```
組み合わせ ロボット，A
  プリント　A
end
```

Aは，ループ処理内部だけで使用できるキャラクタ型の変数で，ループの各処理に先立って，ゲームシーン中に存在する「ロボット」キャラクタが代入される．

次の例は，ゲーム中に存在するすべての「ロボット」キャラクタの，2 つの非順列組み合わせをコンソールに表示する．

```
組み合わせ ロボット，A　B
  プリント　A，B
end
```

前の例と同じく，ループ中は,「作る　ロボット」によりゲーム中に存在する，ロボットキャラクタが順次，AとBに代入されるが，異なるのは，AとBの組が非順列組み合わせとなる点である．例えばゲームシーン中に R1,R2,R3 という 3 つのロボットキャラクタが存在する場合，上記の例によるループ処理は，(A=R1 B=R2)(A=R1 B=R3)(A=R2 B=R3) という 3 回のループとなる．

ループ脱出文

繰り返し処理の記述内で繰り返しを停止して，他の処理の実行を可能にするために，すべてのループ処理内部では,「ループ脱出」文を使いループの処理を強制的に終わらせることができるようにした.

次の例は 10 回のループを無限ループ文を使って記述したものである.

```
回数 = 0
無限ループ
  IF 回数 == 10
    ループ脱出
  END
  回数 = 回数 + 1
終わり
```

3.3.5 時間に沿った処理の記述機能

S540 は,「待つ」命令を使い，一定時間処理を停止する記述をできるようにした．この機能により，時間の経過に従い処理を変化させる記述が可能になっている．また，指定条件が満たされるまで処理を停止させることもできる.

「待つ」命令は，一定時間処理を停止する機能と，指定の条件が成立している間処理を停止する機能の 2 つの機能を持つ.

また，1 フレームだけ「待つ」命令を簡易に記述するために「同期」命令も用意した.

指定フレーム数の時間だけ処理を停止する記述方法

「待つ」命令で指定するフレーム数の間，処理を停止させることができる.

```
待つ 10
```

また，次のフレームまで処理を停止するには「同期」と記述する．次の例ではフレームごとに「テスト」という文字列をコンソールに出力する.

```
無限ループ
  プリント "テスト"
  同期
終わり
```

この例は無限ループ中に毎回「同期」が実行されるので，結果としてフレーム毎に「プリント "テスト"」が実行される.「同期」と「無限ループ」を用いることで，毎フレーム同じ事を繰り返す処理を記述することができ，定形の動作を続けるようなゲームキャラクタの記述が容易となる.

条件成立まで処理を停止する記述方法

「ある一定距離にプレーヤーが近づいてくるまで移動しない」のような動作のために,「待つ」には，与えられた条件が成立している間は処理を停止する機能も用意した．次は，変数「体力」が 10 未満の間処理を停止する．

```
待つ 体力 < 10
```

「待つ」は，次に挙げる条件ループを使った方法で代替可能だが，頻繁な使用が予想されることから，可読性のために用意している．

```
条件ループ 体力 < 10
  同期
終わり
```

3.3.6　並行処理の記述機能

3.2.4 節で述べたように，S540 は，自律的なゲームキャラクタの挙動を記述しやすいよう，定義された状態遷移処理を他のキャラクタとは並行に実行するモデルとし，1 つのキャラクタに 2 つ以上の並行する状態遷移処理を記述できる機能も用意した．

全ての並行処理は，その内部に複数の状態を持ち，並行処理内部の状態間で遷移することができるようにしている．

複数のキャラクタによる並行処理

次は,「あるキャラクタ」を 3 つ「作る」によって出現させる例である．

```
作る あるキャラクタ
作る あるキャラクタ
作る あるキャラクタ
```

出現したキャラクタはそれぞれ定義されている処理の自律的な実行を開始する．複数のキャラクタを出現させると，それぞれのキャラクタの処理が並行に実行される．

1 つのキャラクタ内での並行処理

次は，1 つのキャラクタに 3 つの並行する処理を記述した例である．

```
あるキャラクタ
  @状態 1
    # 状態 1 の処理
  =========
  @状態 2
    # 状態 2 の処理
  =========
  @状態 3
    # 状態 3 の処理
終わり
```

「=======」でキャラクタ内部を区切ることで，複数の並行する処理を記述できる．この区切り記号は=を 1 行に 3 つ以上並べる．3 つ以上であれば，可読性のためにいくつ並べても良い規則とした．

上記の例の場合,「@状態 1」と「@状態 2」と「@状態 3」は，独立した処理の流れに属し，それぞれ自律して処理が実行される．この「=======」で区切られた状態の集まりを「スレッド」と呼ぶ．S540 では，自律的な動作はすべてスレッドである.「=======」が無いキャラクタは，スレッドが 1 つだけしかないキャラクタである．

スレッドは独立した実行単位であるので，異なるスレッドに属する状態への遷移はできない．例えば次の記述は,「@状態 2」が「@状態 1」とは異なるスレッドに属するため,「GOTO @状態 1」がエラーとなる．

```
あるキャラクタ
  @状態 1
  =========
  @状態 2
    GOTO @状態 1
終わり
```

並行処理を利用した状態の変更

同じキャラクタに属するスレッドであれば，他のスレッドの状態でも変更することができる．次の例は,「@初期状態」から「@イベント発生処理」へ状態を変更するが，状態の変更は,「@イベント監視処理」のあるスレッドにより行われる．

```
あるキャラクタ
  @初期状態
  @イベント発生処理
  =========
  @イベント監視処理
    条件ループ …
      CHANGE @イベント発生処理
      同期
    END
終わり
```

この機能により，キャラクタの動作処理とは非同期に発生するゲーム中のイベントをきっかけとした状態遷移を記述することができる．

3.3.7　並行処理に適した通信機能

S540 が提供する通信機能は，2 章で述べた Tuple Space を採用した．これは，次の理由による．

- 様々な通信を表現できる．
- 通信による制御の流れの複雑化を回避できる．
- シンプルな構造と操作は理解しやすい．

本節では，S540 における Tuple Space の操作方法を説明する．

Tuple Space への書き込み

次は，Tuple Space へメッセージを書き込む例である．

```
書く（メッセージA，１，３）
書く（メッセージB，"abcde"）
```

「書く」は，Tuple Space へ Tuple を書き込む命令である．「書く」の第 1 引数は Tuple の名前で，Tuple Space 上で Tuple を区別するために使用される．Tuple の名前以降は，「，(カンマ)」[3]で区切って，引数として任意の値を任意個数だけ持たせることができる．

Tuple の書き込みに関して重複検査は行われない．よって，まったく同じ Tuple を Tuple Space に複数回書き込むと，同じ内容の Tuple が複数存在することになる．

「書く」は英字で “out” と書くこともできる．

[3] 3.3.2 節で述べたとおり，カンマは「，(全角カンマ)」,「,(半角カンマ)」,「、(読点)」のどれでもよい

Tuple Space からの読み込み

Tuple Space からの読み込みは，Tuple の名前と引数を組にして Tuple Space から Tuple の検索を行う．

Tuple Space から読み込む方法には次の 2 つの方法がある．

- Tuple Space から読んで，読んだ Tuple を消す．
- Tuple Space から読むが，読んだ Tuple はそのまま残す．

読み込み後 Tuple を消す命令を「取る」，そのまま Tuple Space に残す命令を「読む」として用意した．

「取る」「読む」の両者とも,「書く」と同じく名前と引数を取り，指定された名前と引数がすべて一致する Tuple を，Tuple Space から検索する．検索の結果一致した Tuple は読み込み処理される．読むべき Tuple が無い場合は,「待つ」命令 (3.3.5 節) と同様に処理を停止し，他の処理から該当する Tuple が書き込まれるまで処理を停止する．

次の例は，名前が「メッセージA」で 2 つの引数「1」と「3」からなる Tuple を，Tuple Space 内から検索し，あればその Tuple を Tuple Space から消す．「メッセージA，1，3」という Tuple がなければ，他の処理からこの Tuple が書き込まれるまで処理を停止する．

```
取る（メッセージA，1，3）
```

「取る」を「読む」にすることで，Tuple Space から Tuple を取り除かずに Tuple を読み込む処理となる．

「取る」は英字で “in” と書くこともできる．また,「読む」は英字で “rd” と書くこともできる．

処理を停止しない Tuple Space からの読み込み

ある Tuple があるかどうかが知りたい場合，読み込み時に処理が停止するよりも，真偽値で検索の結果を知ることができるほうがよい．そこで,「取ってみる」と「読んでみる」という命令を用意している．これらはいずれも Tuple があれば「取る」「読む」と同様に動作するが，Tuple が無い場合は待ち状態に入らずに終了する．Tuple があったか否かは命令の返す真偽値で確認できる．次は「取ってみる」の使用例である．

```
IF  取ってみる（メッセージA，1，3）
  プリント  "取れました"
終わり
```

ある Tuple が，Tuple Space に存在し続ける間処理を繰り返す場合は，次の記述をする．

```
条件ループ　取ってみる（処理スイッチ）
  プリント　"処理をする"
終わり
```

上記の例は,「処理スイッチ」という，引数のない Tuple が Tuple Space にある間,「処理をする」とコンソールに表示し続ける．この処理を有効にする場合は，次に挙げる，他のスレッドから「処理スイッチ」の書き込み処理を行う．

```
書く（処理スイッチ）
```

逆に，次の例を実行することで，処理を停止させることができる．

```
取る（処理スイッチ）
```

「取ってみる」は英字で “inp”,「読んでみる」は英字で “rdp” と書くこともできる．

匿名引数を使用した読み込み

読み込みの処理は，名前と引数がすべて一致する Tuple について行われるが，匿名引数にすることで，値を指定せず任意の値の Tuple を読み込むことができる．次の例は名前は「メッセージ」，第一引数は「5」，第 2 引数は任意のデータの Tuple を読み込む記述である.「?」は匿名引数を意味する．

```
取る（メッセージ，5，?）
```

匿名引数はデータ型を指定することもできる．これにより，特定の型の引数を伴う Tuple だけを読み込むことができる．次の例は前の例と違い，第 2 引数が整数データの Tuple を読み込む.

```
取る（メッセージ，5，?:整数）
```

引数値を変数へ代入する読み込み

匿名引数に対応する引数値を必要とする場合，匿名引数の代わりに，引数値を代入する変数を指定することができる．前節の例を次の例に変更することで，読み込んだ Tuple の第 2 引数の値が「データ」という変数に代入される．

```
データ ： 整数
取る（メッセージ，5，?データ）
```

この例は変数の宣言と「取る」命令が分かれているが，これを 1 つにまとめて単純に記述する形式も用意されている．

```
取る（メッセージ，5，?データ:整数）
```

Tuple の引数値を取得する機能を利用することで，様々な通信が可能となる．次の例は，2 つの整数の和を出力する機能を実装する記述である．

```
データ 1：整数
データ 2：整数
無限ループ
  取る（加算処理，?データ 1，?データ 2）
  プリント データ 1 + データ 2
終わり
```

このプログラムが実行されている状態で，次の例の，整数の引数を 2 つとる「加算処理」という名前の Tuple を Tuple Space へ書き込む処理を行うと，2 つの引数の加算結果がコンソールに表示される．

```
書く（加算処理，10，20）
```

3.3.8 簡単なプログラム例

本節では簡単なプログラム例を通して，S540 のゲームキャラクタの記述方法について説明し, 他のプログラミング言語との比較を行う．

図 3.4 と**図 3.5** は「ロボット」という名前のキャラクタの記述例で，**図 3.3** はこのプログラム例の構造図である．

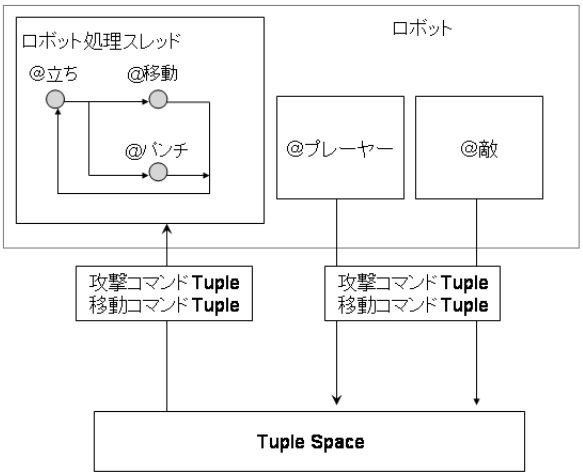

図 3.3: **図 3.4** と**図 3.5** の構造図

「ロボット」には 3 つの並行処理が記述されている．1 つは，**図 3.4** の部分で，歩いて移動したりパンチというアクションの処理を行い，残りの 2 つ

```
ロボット

  @開始
    人間 ("human_model")
    GOTO  @立ち

  @立ち
    リピートモーション ("idle_motion")
    無限ループ
      IF 取ってみる (攻撃コマンド, 自分)
        GOTO @パンチ
      ELIF 読んでみる (移動コマンド, 自分,?)
        GOTO @歩く
      END
      同期
    終わり

  @パンチ
    モーション ("attack_motion")
    待つ モーション再生中 ()
    GOTO @立ち

  @歩く
    リピートモーション ("walk_motion")
    条件ループ 読んでみる (移動コマンド, 自分, ?方向:実数)
      振り向く (方向,360)
      進む (15)
      同期
    終わり
    GOTO @立ち
```

図 3.4: 移動や攻撃の処理をするロボットのプログラム (前半)

は，**図 3.5** の部分で,「ロボット」をコントロールする部分である．ロボットのコントロール方法には，レバーやボタン等のプレーヤーの入力を使う方法と，移動やパンチを適当な間隔で繰り返す敵キャラクタの処理の 2 つがあり，異なる並行処理として記述されている．

ライブラリ

図 3.4 と**図 3.5** の例は，**表 3.3** のアプリケーション下位層の機能を提供するライブラリが，すでに実装されているものとして記述されている．ライブラリは画面に 3D モデルを表示しアニメーションをさせるグラフィクスの機能と，プレーヤーの入力を読む外部機器の状態入力機能の 2 つの機能を簡単に実装している．

グラフィクス機能は表示する 3D モデルデータを設定する「人間」と，その 3D モデル上で任意のアニメーションデータを再生する「モーション」「リ

```
================
@プレーヤー
  取る(プレーヤー開始,自分)
  無限ループ
    IF レバー()
      書く(移動コマンド,自分,レバー向き())
    ELIF レバーオフ()
      取ってみる(移動コマンド,自分,レバー向き())
    END
    IF ボタンオン(0)
      書く(攻撃コマンド,自分)
    ELSE
      取ってみる(攻撃コマンド,自分)
    END
    同期
  終わり

================
@敵
  取る(敵開始,自分)
  無限ループ
    ループ 4
      書く(移動コマンド,自分,乱数(0,360))
      待つ 60
      取ってみる(移動コマンド,自分,?)
    終わり
    書く(攻撃コマンド,自分)
    待つ 60
  終わり

終わり # ロボットの定義終わり
```

図 3.5: ロボットを操作するプログラム (**図 3.4** の続き)

ピートモーション」，アニメーションの再生状態を調べる「モーション再生中」の 4 つのライブラリ手続きで実装されている．

外部機器の入力機能は，プレーヤーがキャラクタを移動させるのに使用するレバー状態を調べる 3 つのライブラリ手続きと，ボタン状態を調べる手続きからなっている．

S540 記述ツールが提供するライブラリ機能の詳細については 3.4.2 節で述べる．

キャラクタの動作の記述

図 3.4 は次の 4 つの状態を持つ「ロボット」という名前のキャラクタの記述例である．

- @開始．キャラクタの初期化をする．

表 3.3: **図 3.4**，**図 3.5** で使用しているライブラリ手続き

人間	指定の表示 3D モデルを設定
モーション	指定のアニメーションの再生を指示する
リピートモーション	指定のアニメーションの繰り返し再生の指示
モーション再生中	アニメーション再生が終わっていなければ「真」を返す
振り向く	キャラクタを指定方向へ回転する処理を 1 フレーム分だけ行う
進む	キャラクタを指定の速さで前進する処理を 1 フレーム分だけ行う
レバー	入力機器のレバーがいずれかの方向に倒されていれば真を返す
レバーオフ	レバーが倒されなくなったら真を返す
レバー向き	レバーが倒されてる方向を返す
ボタンオン	指定番号のボタンが押されたなら真を返す

- @立ち．キャラクタが立っているだけの状態.
- @パンチ．キャラクタが攻撃技を出している状態.
- @歩く．キャラクタが移動をしている状態.

「@開始」以外の各状態は，Tuple Space からキャラクタ自身に対するメッセージを受け取って，他の状態へ遷移を行う.

例えば「@立ち」状態では，次に挙げる遷移処理を行う.

- Tuple Space に「攻撃コマンド」というメッセージがあれば「@パンチ」へ遷移
- Tuple Space に「移動コマンド」というメッセージがあれば「@歩く」へ遷移

この処理は無限ループにより繰り返されておりメッセージが見つからない間は「@立ち」状態が続く．この無限ループの処理の前に,「リピートモーション」により，”`idle_motion`”という立っている状態のアニメーションの繰り返し再生が指定されているので,「@立ち」状態の間キャラクタは立っているアニメーションが再生され続ける.

「@パンチ」状態は,「@立ち」で「攻撃コマンド」という名前の Tuple が, Tuple Space に書き込まれることで遷移してくる状態である.「攻撃コマンド」の引数が「自分」であるものの場合のみ遷移するので，他に同じ「ロボット」のキャラクタが「作る」により存在しても，他のキャラクタの通信と混同されることがない.

「@パンチ」状態は，パンチ動作アニメーション”attack_motion”の再生が指定されたあと,「モーション再生中」が真の間処理を停止し，その後「@立ち」状態へ遷移する．これにより，パンチ動作のアニメーションが終了するまで「@パンチ」状態は継続し，パンチ動作が終了後「@立ち」へ状態が戻る．

「@歩く」状態は”walk_motion”という名前の歩き動作のアニメーションを再生後,「移動コマンド」という名前の Tuple が Tuple Space にある間，移動処理を無限ループで実行し続ける.「移動コマンド」には「@パンチ」と同様，引数には他のキャラクタへのメッセージと区別するための「自分」があるが，もうひとつ，移動する方向も持つ.「@歩く」の動作処理中は常に Tuple Space 上の「移動コマンド」という書き込みを確認し，引数で指定される方向へキャラクタを移動させる処理を続けることで，キャラクタは「移動コマンド」の指定する方向へ移動し続ける．

キャラクタコントロール部分の記述

図 3.5 は**図 3.4** から続いている記述であり，次の 2 つが記述されている．

- プレーヤーによるロボットのコントロール処理
- 簡単な戦略を実装したロボットの自動コントロール処理

二つの処理は並行処理として**図 3.4** のキャラクタの動作とは独立した処理として記述されている．

「@プレーヤー」という状態を持つほうのスレッドは，レバーやボタン入力にあわせて「ロボット」に対するコマンドを Tuple Space へ書き込む処理を行う．同様に「@敵」という状態を持つほうのスレッドは，レバーやボタン入力の代わりに，適当な時間，適当なタイミングで，移動コマンドや攻撃コマンドを Tuple Space へ書き込む処理を行う．

どちらのスレッドも，起動した直後に Tuple Space から「プレーヤー開始」もしくは「敵開始」という Tuple を取りにいく．該当する Tuple が無ければ処理が開始されないので，この Tuple は処理の開始スイッチとして役割を持つことになる．例えば，2 体のロボットというキャラクタを出現させ，片方はプレーヤーによるコントロール，片方は非プレーヤーの自動的なコントロールとしたければ次の記述をする．

```
書く（プレーヤー処理，作る　ロボット）
書く（敵処理，作る　ロボット）
```

3.3.9 汎用言語との比較

この節では，S540 で記述したものと，他の汎用プログラミング言語で記述したものを比較して，S540 との相違点を述べる．

比較には，プログラミング言語 Java[Gos00] を選択した．Java は，クラスをベースとした汎用のオブジェクト指向言語であり，現在のビデオゲーム開発で主に用いられる C++ 言語と基本的な構造が同等で構文も類似している．このため，Java プログラムとの比較は，現在のゲーム開発における記述との比較とすることができる．また，Java には，並行処理のためのスレッド以外に 1.9 節の要件を満たす機能を持たないので，各機能について S540 がどの程度記述性を改善しているのかの比較もしやすい．

図 3.6 は，S540 で記述した**図 3.4** のロボットの処理部分と同様の処理を，汎用プログラミング言語である Java で記述したものである．ライブラリで提供される「人間」や「モーション再生」は，「`model(String name)`」「`playMotion(String name)`」として `Character` クラス中で定義済みであるとする．Java プログラムは S540 で記述したものと比較して次の特徴がある．

- 並行処理の記述にスレッド機能を用いている．
- メッセージの受信処理が動作の処理と分かれている．
- 状態の識別に変数 (`state`) や定数 (`IDLE`,`PUNCH` など) を用いている．

Java では，Thread クラスや Runnable インターフェースを用いて並行処理を記述することが可能である．この方法で並行処理を記述する場合，キャラクタの動作処理は `run()` メソッドに記述することになる．

Java には，Tuple Space やそれに近い通信機能がないため，通信の記述は各クラスに定義されるメソッド呼び出しを用いる．このため受信処理を動作処理とは別のメソッドとしなければならない．

また Java には，明示的に状態遷移処理を記述する方法が用意されていないことから，状態を記憶する変数 `state` や状態の ID を定義する定数 `IDLE`,`PUNCH`,`WALK` が必要となる．

この Java による記述は S540 と比較して次の問題点がある．

- 状態遷移処理の記述が簡単ではない．
- プログラムが断片化し記述が複雑になりやすい．
- 処理の流れが複雑になりやすい．
- 不要なデータ管理をしなければならない．

Java では，状態遷移処理の記述や状態の管理のために，変数や条件分岐を必要とするので，1.9 節の**図 1.4** と同等のプログラムとなっている．そのため，状態遷移処理の記述は見通しが悪く，間違いの起こりやすいものになっている．

受信処理を，キャラクタの動作処理と分けて記述しなければならないことから，受信処理のメソッドが多く必要になりプログラムは断片化しやすい．

受信処理メソッドを用いた通信は，キャラクタ内部で定義される処理とは異なる処理の流れがキャラクタの内部のメソッドにも渡るため，記述者は複

```
public class Robot extends Character implements Runnable {
  /* 状態を表す ID */
  static final int IDLE = 0;  /* 立ち状態 */
  static final int PUNCH = 1; /* パンチ状態 */
  static final int WALK = 2;  /* 歩き状態 */
  int state = IDLE, dir;
  /* 初期化 */
  public Robot() { model("human_model"); }
  /* 処理の委譲 */
  void sync() { /* ... */ }
  /* メッセージ受信 */
  synchronized public void punch() {
    if (state == IDLE) state = PUNCH;
  }
  /* walk コマンドの処理 */
  synchronized public void walk(int dir) {
    if (state == IDLE || state == WALK) {
      state = WALK; this.dir = dir;
    }
  }
  /* 動作処理 */
  public void run() {
    for (;;) {
      switch (state) {
      /* 立ち状態の処理 */
      case IDLE:
        repeatMotion("IDLE_motion");
        while (state == IDLE) sync(); break;
      /* パンチ状態の処理 */
      case PUNCH:
        playMotion("attack_motion");
        while (!isEndOfMotion()) sync();
        state = IDLE; break;
      /* 歩き状態の処理 */
      case WALK:
        repeatMotion("run_motion");
        while (state == WALK) {
          turn(dir);
          advance(15);
          state = IDLE;
          sync(); }
        state = IDLE; break;
      }
    }
  }
  public static void create() {
    (new Thread(new Robot())).start();
  }
};
```

図 3.6: Java で記述したロボットの例

雑な処理の流れを管理しなければならない．また，Java のスレッドは処理のスケジューリングやデータ競合の解決が難しいことから，ゲームキャラクタの処理を実装するには適切でないこともある．その場合，フレーム毎に呼び出して処理を進めるメソッドを `run()` メソッドの代わりに定義しなければならず，プログラム全体の処理の流れはさらに複雑になる．

キャラクタの動作処理と通信の受信処理が異なるため，状態の変更処理や，動作処理への受信データの受け渡しは，受信処理側で行わなければならない．このため，動作処理と受信処理メソッド間で共有して使用できるデータが必要となり，これらの共有データの管理も行わなければならない．

これらのことから，S540 は Java と比較して，ゲームシステムの記述に必要な状態遷移処理や並行処理，通信の記述に優れている．

3.4　S540 のゲームアプリケーションへの組み込みと実装

3.4.1　組み込み時のゲームアプリケーションの構造

S540 記述ツールは様々なゲームアプリケーションへ組み込んで使用することを想定している．そのため，特定のゲームアプリケーションに特化しない構造と実装にした．

アプリケーションへの組み込みは，S540 で記述されたゲームシステムプログラムをトランスレーターにより変換した C++プログラムと，S540 記述ツールが提供する S540 プログラム実行系を，アプリケーションプログラムとまとめてリンクすることで行う．

S540 は，ゲームシステムの記述性のためのゲーム記述に特化したプログラミング言語であるが，個々のビデオゲーム製品に特化した機能は提供しない．代わりに，様々な製品に組み込みやすい構造をアプリケーションに提供し，組み込んだアプリケーション側で必要な機能を追加する構造となっている．

図 3.7 は，S540 記述ツールを用いてゲームアプリケーションを開発する場合の構造図である．ゲームデザイナーにより記述されたゲームシステムプログラムは，グラフィクスやハードウェアコントロールを行うアプリケーション下位層の上位に位置する．ゲームシステムプログラムとアプリケーション下位層の間には，S540 プログラム実行系があり，両者をつなぐインターフェースの役割や，記述ツールが提供する機能のうち実行時に必要なものを提供している．

この構造により，記述ツールによる記述部分はアプリケーションでもっとも上位層となり，アプリケーションの動作全体，すなわちゲームシステム全体が記述ツールを用いて記述可能となっている．また，S540 プログラム実行系は個々の製品のアプリケーション下位層とは分離できる設計となっており，記述ツールを様々な製品開発に利用しやすい構造となっている．

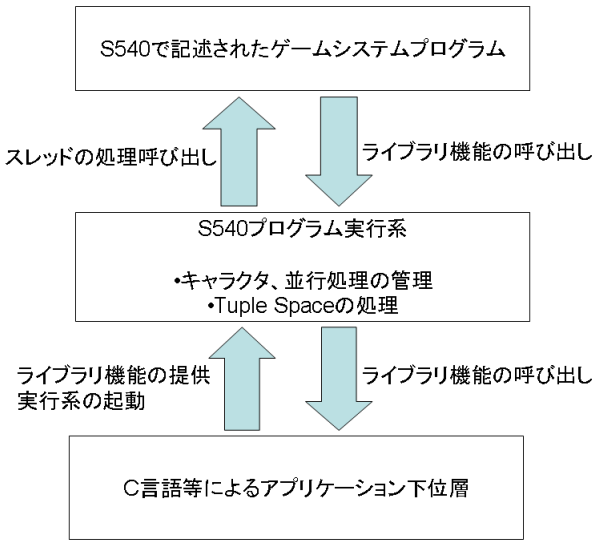

図 3.7: S540 記述ツールを使用したゲームアプリケーションの構造図

3.4.2 ライブラリ機能

アプリケーション下位層の開発はプログラマーに任されており S540 記述ツールでは提供されない.

ゲームデザイナーが必要とするグラフィクスやサウンド等のアプリケーション下位層の機能は，下位層のプログラマーが 3.3.8 節で述べたライブラリとして記述ツールに提供する.

S540 記述ツールの提供するライブラリ機能は次の特徴を持つ.

- ライブラリは全て手続きとして登録される
- 「待つ」命令のような処理を停止する機能は持たせられない

3.3.8 節の「人間」や「振り向く」等のライブラリ手続きの実装をする場合，ライブラリ実装を行うプログラマーは次のプログラムを作り，実行時に C++ 関数を呼び出す指示を，トランスレーターに対して行う.

```
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 言語プログラムの直後の () で囲まれた部分には，呼び出すときの引数の S540 のデータ型を指定する．次の例は，これら 2 つのライブラリの C 言語プログラム側の実装の一部である.

```
// 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
}
```

C 言語プログラム側の関数は必ず次の引数型としなければならない.

```
(SObj* obj, Tuple* arg, void* res)
```

ゲームシステムプログラム側から渡される実際の引数を取得するには，Tuple 型の第 2 引数 arg の iterator() 関数で反復子を取得し，その反復子のデータ型に応じた引数取得関数 get_int() や get_string() や，反復子を次の引数に進める next() 関数を用いて行う.

ライブラリが値を返す場合は，第 3 引数の res ポインタの指す場所に戻り値を設定する．次は 2 整数の加算を行う関数の S540 ゲームシステムプログラム用の記述である.

```
extern int 足す "add" (int, int)
```

この S540 の手続きに対する C 言語プログラム側の実装は次である．res を使って加算結果をゲームシステムプログラム側へ返している.

```
void add(SObj* obj, Tuple* arg, void* res) {
  Tuple::iterator i = arg->iterator();
  int a = i.get_int();  i.next();
  int b = i.get_int();
  *((int*)res) = a + b;
}
```

この「足す」手続きは，次のようにゲームシステムプログラム中で使用できる.

```
結果：整数
結果 = 足す（1, 5）
```

3.4.3　S540 記述ツールの実装

S540 記述ツールは次のものから構成されている.

- ゲームシステムプログラムを記述したテキストファイルを入力とするトランスレーター
- ゲームプログラム本体に組み込むための S540 プログラム実行系

トランスレーターはゲームデザイナーがゲームシステムを記述したテキストファイルを入力し，ゲームプログラム本体と結合できる C 言語プログラムへ変換する．

S540 プログラム実行系もゲームプログラム本体と結合されるプログラムであり，記述ツールの機能の中でゲームプログラム実行時に処理する必要のある機能を実装する．

図 3.8 は S540 記述ツールを用いた場合の記述データの流れをあらわしたものである．ゲームデザイナーにより，テキストファイルとして記述されたゲームシステムプログラムは，トランスレーターにより C 言語プログラムへ変換される．変換後の C 言語プログラムは，ゲームアプリケーション本体のプログラムと一緒に，C 言語コンパイラによりゲームの実行形式となる．ゲームアプリケーション本体のプログラムには S540 プログラム実行系が含まれる．

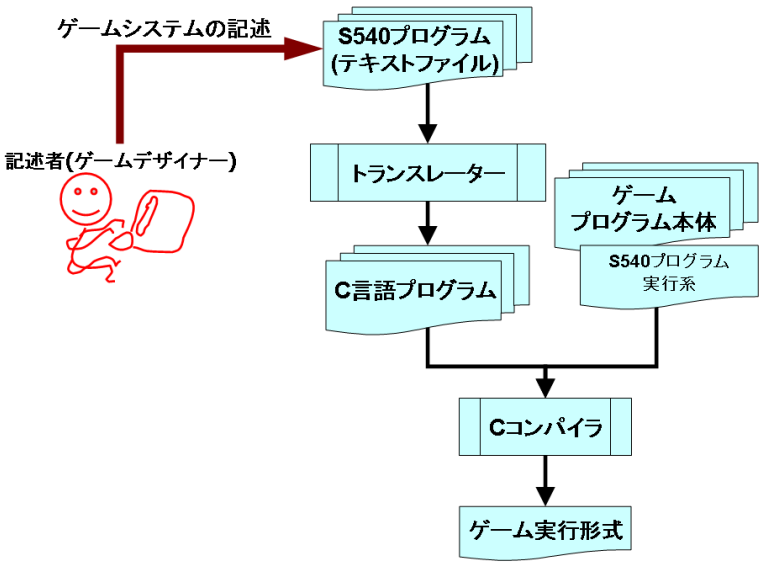

図 3.8: S540 記述ツールを使用した記述データの流れ

トランスレーター

トランスレーターはゲームシステムプログラムが記述されたテキストファイルを入力し，ゲームプログラムへ組み込むことで実行可能な C 言語プログラムを出力する．トランスレータープログラムの規模は 8500 行程度で最初の

バージョンは 2ヶ月程度で完成した．3.5 章で述べる開発現場へ導入を開始した後も，現場からの要求に応じて修正を行った．

トランスレーターは次の実装となっている．

- 全て Java で標準的なライブラリのみを用いて実装
- 構文解析プログラム生成系に SableCC を使用

トランスレーターの実装には Java を使用しており，入力テキスト読み込み処理プログラムを生成する構文解析プログラム生成系も Java のみで実装された SableCC[GH98] を使用している．Java を使用し，標準的なライブラリのみで実装されたトランスレーターであるため次の特徴を持つ．

- 日本語文字コードを入力プログラムに使用可能
- 複数の OS 上でも動作

Java の入出力ライブラリは多バイト文字に対応しているため，トランスレーターの入力には日本語文字が利用可能となっている．また，Java で実装されたプログラムは再コンパイルなしに複数の OS 上で動作するので，トランスレーターは複数の OS 上で動作する．

S540 プログラム実行系

S540 プログラム実行系は様々なビデオゲーム開発に用いられることを想定し，アセンブラや特定のプラットフォームに依存した機能を用いない標準的な C++言語だけで実装されている．

図 3.7 のように，S540 プログラム実行系はアプリケーションの中間層に位置し，S540 で記述されたゲームシステムプログラムとアプリケーション下位層とをつなぐ役割を果たす．S540 プログラム実行系は，次のプログラムで構成されている．

- キャラクタや並行処理の管理
- Tuple Space の処理
- ゲームシステムプログラムとアプリケーション下位層とのインターフェース

ゲームシステム中のキャラクタやスレッドの管理については，アプリケーション下位層のプログラムが実装を行う必要はない．これにより，プログラマーは記述ツールに対するライブラリを提供することに専念できるので，ゲームシステム開発と下位層の開発を並行作業とすることができ，開発作業の効率化が期待できる．

同様に，実行時の Tuple Space の処理も，記述ツールが S540 プログラム実行系で提供するので，下位層のプログラマーは意識する必要がない．

ゲームシステムプログラムは，3.4.2 節で述べたライブラリ機能を通して，アプリケーション下位層の機能を呼び出しすことで，提供される機能を使用する．このとき，中間層である実行系は，3.4.2 節のライブラリ機能で指定された適切なアプリケーション下位層のプログラムを呼び出し，ライブラリ機能で結果を返すことが定義されているならばゲームシステムプログラム側に実行結果を返す．

3.5 実際の開発プロジェクトへの適用と評価

3.5.1 評価プロジェクトの概要

開発した S540 の有効性を評価するために，実際のビデオゲーム開発現場で使用し，製品となる前段階の，プロトタイプのビデオゲームソフトウェアの制作を行った．

また，このプロトタイプはそのまま家庭用ゲーム「グラディエーターロードトゥフリーダム」[アー 05] の開発につながった．グラディエーターロードトゥフリーダムは，複数の人間が武器や防具を装備して戦うアクションゲームである (**図 3.9**)．キャラクターの技のバリエーションや装備の組み合わせが豊富で，高度に複雑な戦闘を楽しむことができる．また，戦闘以外にも，複数に分岐するシナリオが容易されており，遊び方によって，異なるストーリーを楽しむこともできるゲームとなっている．

本章では，プロトタイププロジェクトにおける S540 の利用の状況とその評価について述べる．

図 3.9: グラディエーターロードトゥフリーダムの画面

3.5.2　開発規模と開発期間

S540 記述ツールの使用を行った開発会社は，ビデオゲーム開発を目的として，ビデオゲーム開発経験者が数名集まって創立された企業である． 企業としての開発経歴はなかったが，開発者はそれぞれ様々なビデオゲーム関連企業で開発の業務に携わってきていたので，研修や教育なしに開発業務を直ちに開始できる状態であった．

プロトタイプの開発規模や期間は次のとおりである．

- プロジェクト全体では 10 人程度で開発
- 記述ツールで記述作業をしたのはゲーム開発経験者 3 人
- 開発期間はおよそ 4ヶ月

プロトタイプ開発は 10 人前後で行われた．

そのうち S540 記述ツールで記述作業を行ったのはゲームデザイナーの 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 文を使った処理の流れの制御について基本的な知識を持っていたが，もう 1 人は S540 が初めて体験するプログラミング言語であった．

3 人とも商用のゲーム開発の経験があり，過去の開発中では，アイディア検討や新しいゲーム要素の分析や設計をしたり，数値テーブルの記述や修正等の作業を行っていた．しかし，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．

3.5.3　開発ゲームの内容

プロトタイプ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，S540 記述ツール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．**図 3.10** [4] はプロトタイプゲームのスナップショットである．

完成したプロトタイプゲームは次の特徴を持つ．

- 複数の人間同士が戦うアクションゲームである．
- ゲームはキャラクタ選択，プレイモード，ゲームオーバー画面の 3 モードからなる．
- 人間は武器や盾を持ち様々な技を駆使する．装備品により攻撃力や技が変化する．
- プレーヤー以外の人間はプレーヤーを倒すことを目的に動作する．

[4] 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 グラフィクスを使用したものとなっている．

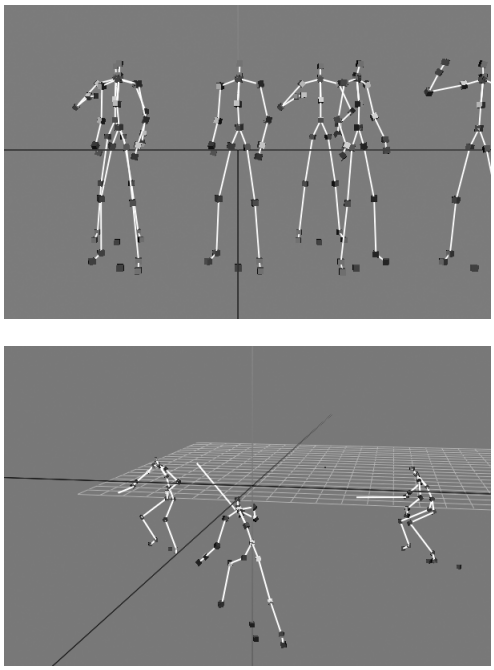

図 3.10: キャラクタ選択中 (左) とプレイ中 (右) の画面 (開発途中)

- ゲームステージ中には柱等の障害物があり，破壊したり攻撃に利用することができる．
- 馬車には乗り込むことができ，移動や攻撃方法が変化する．
- ビジュアルエフェクト等の演出や，現在の体力値や攻撃力の表示機能がある．

開発したゲームは複数の人間同士が武器をもって戦う格闘タイプのアクションゲームで，プレーヤー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，逆にプレーヤーが敵に倒されるとゲームオーバーとなり，キャラクタ選択画面へ戻る．

ゲーム中に登場する人間は様々な技を駆使する．武器や盾を捨てることで身軽になったり，逆に，倒した相手から奪うことで，攻撃力を高めたり新しい技を会得することもできる．

プレーヤーが操作する人間はプレーヤーの入力操作により動作するが，プレーヤー以外の人間はプレーヤーを倒すことを目的とした AI [5] プログラムによりコントロールされる．

[5] ビデオゲーム業界やゲームプログラミングにおいて，プレーヤー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．

ゲームステージ中に配置されている柱などのオブジェクトは人間により切り倒すことができる．切り倒すことで敵に当ててダメージを負わせたり，より戦略的に障害物を作り出したりすることができる．

また，ステージよっては馬車もあり，馬車に乗り込むことで高速に移動したり，強力な攻撃を出すことが可能になる．

ゲーム画面中には上記の他に，攻撃や衝突等を演出するためのビジュアルエフェクトや，プレーヤーがプレイ中の残り体力値や攻撃力等を知るためのゲージ表示等がある．

3.5.4　開発の進行状況

開発は, 導入時期に S540 の学習のために試行錯誤が続いた後，目的とするゲームシステムの記述が始まり，その後，ゲームデザイナーが実装を直接主導する形で進むようになった．以下に開発の進め方を時間順に段階分けして説明する

S540 記述ツールの導入時期

最初に開発者にプログラミング言語 S540 と S540 記述ツールの使い方について学んでもらうことが必要であったことから，記述ツールの導入は次のように行った．

- 入門用テキストを読んでもらいサンプルプログラムを動かしてもらう
- 実際に記述を行いわからないところを随時質問してもらう

記述ツールを用いた開発は，入門用のテキストと，簡単な例として作った**図 3.4**，**図 3.5** のプログラムを，ゲームデザイナーに提供することで開始した．また，わからないところは随時質問してもらった．本論文の末尾 A.2 節に，この導入時の入門用のテキストを付録として掲載している．

S540 記述ツールはゲームシステムのすべてを記述できることを目的としていることから，ゲームデザイナーはゲームシステムのほぼすべてについての実装をしなければならない．このため，ゲームデザイナーには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．

開発中期

導入から 1ヶ月程度でゲームデザイナーは次のことができるようになった．

- 簡単なプログラム記述
- 意図するゲームを開発するために必要なライブラリ機能の認識

- 開発作業の分担

導入時期の試行錯誤である程度コツがつかめてくると，攻撃や防御を使い分ける簡単なプログラムを記述するようになった．これにより，意図するゲームを作るために必要な機能 (ライブラリ) も認識できるようになり，それらの追加や修正の意見を，プログラマーに対して出すようになった．

また，ゲームデザイナー 3 人の間で「ゲームの進行管理」「人間」[6]「エフェクトや AI」という分担ができ，それぞれ自分の記述部分は独自に進め，時々全員の記述を合体しながら開発の進み具合を確認する開発形態となった．各ゲームデザイナーの記述したプログラム同士の通信は Tuple Space を用いて記述された．

開発後期

プロトタイプ開発の後半時期は次のようにゲームデザイナー主導の開発形態となった．

- プログラマーはゲームシステムに関与しなかった
- 開発環境もある程度ゲームデザイナーが制作するようになった

ゲームシステムがゲームデザイナーにより記述されるようになったことから，ゲームシステムに関してはプログラマーはまったく関与せず，ゲームデザイナーからの機能の追加修正要求をこなしていく，という進行体制となった．

また，グラフィクスやアニメーションデータを開発ターゲット機上で確認するツールも S540 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．

3.5.5 開発中に記述されたプログラムの調査

プロトタイプ開発における、S540 の有効性を評価するために、記述されたプログラムの調査を行った．本節では、プログラムを調査した結果について，開発中のゲームデザイナーらの利用状況も加えて説明する．

キャラクタの記述

表 3.4 に開発で作られたプログラムの規模を示す．キャラクタの種類数に比較してプログラムの行数が多いが, これは言語に手続きの定義機能がなく，ほぼ同じ処理をプログラムの複数の場所で繰り返し記述している部分が多いことが原因である．

表 3.5 は記述された主なキャラクタについてまとめたものである．キャラクタの用途は次の 3 つに分類することができる．

[6] ゲーム中に登場する人間キャラクタのこと．

表 3.4: プログラム規模

行数	8910 行 (コメント，空行含む)
キャラクタの種類数	32
1 キャラクタあたりのスレッド数	2 つのキャラクタが 8．残りは 1 つ
状態数	平均約 5．最大 21(人間)

表 3.5: キャラクタの記述内容

キャラクタの種類	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．
ゲームシステム上必要な処理	ゲーム中には登場しないが，キャラクタとして記述されたもの．プレーヤーの入力もしくは敵キャラクタの AI 処理の結果を人間にコマンドとしてメッセージ送信する処理や衝突判定の処理．
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等．ゲームシステムに直接関わらないが，ゲーム中のイベントに応じて動作したり登場するキャラクタ．
ゲームシステムに関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，ゲームの進行管理．

- ゲームシーンに登場するキャラクタの記述
- ゲームシーンに登場しない管理処理等の処理の記述
- 作業分担の単位

人間や馬車，ビジュアルエフェクトはゲームシーン中に目に見える形で登場するキャラクタである．これらのキャラクタは記述中でも 1 つのキャラクタとして記述されていた．人間には「立ち」や「攻撃」等の多くの状態が記述され，馬車も「誰も乗っていない」状態と「人間が乗って操作されている状態」等の状態があり，ゲーム中はこれらの状態を遷移しながらキャラクタの処理が進むようになっていた．

カメラやゲームシーンの照明は，ゲームシーン中では見えないが，コンピューターグラフィクスにおいては独立したオブジェクトとして扱うことが一般的であることから，ゲームデザイナーはこれらを見えるキャラクタと同様にキャラクタとして記述していた．また，敵キャラクタの AI，画面効果等

表 3.6: 2 つスレッドを持つキャラクタの処理内容

キャラクタ	各スレッドの処理内容
人間，剣，柱	自身のアクション
	衝突判定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
AI(敵思考) 処理	AI 処理
	キャラクタの状態や位置情報などの思考処理に必要な情報の収集
衝突判定の処理	キャラクタの体同士の排他処理
	攻撃判定処理

の管理，ゲームの進行管理の処理は，ゲーム中に登場するキャラクタではないが，ゲームデザイナーには「見えないキャラクタ」として理解されていたことから，独立したキャラクタとして記述されていた．これらのゲームシーン中に登場しないキャラクタは，状態を 1 つしか持たないものが多かったが，「ゲームの進行管理」については，キャラクタ選択，プレイモード，ゲームオーバーというモードを状態として記述していた．

キャラクタを使うことで，プログラムをより小さい部分に分割できることから，3.5.4 節で述べた記述者間での「ゲームの進行管理」「人間」「エフェクトや AI」という作業分担の単位としてもキャラクタは利用されていた．

並行処理の使われ方

2 つのスレッドが定義されたキャラクタ 6 種について，スレッドの使われ方を**表 3.6** にまとめた．

並行処理の用途としては次の 3 つがあった．

- ある処理とは非同期な状態遷移をする処理の記述
- 独立性が高く自律したものとするほうが良い処理の記述
- 並行処理として記述するとわかりやすくなる処理の記述

並行処理は非同期な状態遷移の記述するに利用されていた．例えば，**表 3.6** 中の「人間」や「馬車」等の処理には,「攻撃が当たるとやられ状態に変化」「人間に乗り込まれると，人間による操作状態に変化」のような，キャラクタの通常の動作処理とは非同期な状態遷移があることから，3.3.6 節の異なるスレッドの状態変更機能を用いて，遷移処理のみ自律した処理として記述されていた．また,「馬車」には「人間」が乗り降りできるが，乗り降りは馬車の

状態を馬車の処理とは非同期に変化させることから，人間と同様に遷移処理のみ自律した処理として記述されていた．

わかりやすさや記述のしやすさから，独立性の高い処理は自律した処理としているものもあった．敵の AI 処理の場合，AI 処理本体と AI 処理に必要な情報の収集は処理内容が全く異なり，独立した処理として記述する方がわかりやすいことから，異なるスレッドとして分けて記述していた．また，衝突判定処理には，キャラクタ同士がお互いにめり込むのを避けるための「排他処理」と，攻撃が相手に当たったかどうかを検査する「攻撃判定処理」の 2 つがあるが，ゲームデザイナーには異なる処理と認識されており，異なるスレッドで記述されていた．

人間は 1 つのキャラクタとして記述されているが，AI やプレーヤーの入力処理は，人間とは別のキャラクタとして記述されていた．人間の本体の処理とコマンドを送る処理は，自律した処理として分けて記述する方が全てを 1 つの処理として記述するよりもプログラムが簡潔になり，わかりやすくなることが 2 つのキャラクタとして記述された理由として挙げられる．また，もう一つの理由として，人間と AI は異なるゲームデザイナーが担当しており，記述が分かれていると分担もしやすいことが挙げられる．

Tuple Space の使われ方

主な Tuple Space の使われ方を**表 3.7** に，実行時の主な Tuple についてのアクセス量を**表 3.8** に示す．

Tuple Space は様々な使われ方をしており，読み書きのタイミングやアクセス量も様々であるが，大きくは次の 3 つの用途に使用されていた．

- 1 つ以上のキャラクタに対する何らかの通信
- 複数の処理やキャラクタ間でのゲーム情報の共有
- 処理間で受け渡すパラメータ

Tuple Space は，あるキャラクタに対して何らかのイベントや処理に応じた処理をさせたいときの通知の手段として多く利用されていた．**表 3.7** では「メッセージ通信」としてまとめた Tuple がこれにあたる．例えば,「コマンド」Tuple は，プレーヤー入力処理や AI 処理から人間に対してアクションをさせるために送られるメッセージで，引数には「移動」や「攻撃」というアクションを表す文字列とアクションに必要なパラメータを含む.「カメラ揺らし」はカメラキャラクタに対するメッセージで，プレーヤーにダメージ等で画面全体を揺らしたいときに，Tuple Space を通してカメラキャラクタに送られる．

表 3.7 の「グローバル変数」,「キャラクタの状態を公開」,「キャラクタ存在フラグ」にある Tuple は，ゲーム中のキャラクタが必要とする情報を保持する Tuple である．例えば,「状態」Tuple は，ゲーム中の人間の位置や体力を

表 3.7: Tuple Space の主な使われ方

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	プレーヤーの選択したキャラクタ名を保持.
	プレーヤーロボット	プレーヤーキャラクタを保持
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.
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 (人間) の状態を保持. 外部から人間の状態を調べるために利用される. 人間の数だけ Tuple Space 上にある.
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	人間に「移動」「攻撃」など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	衝突判定処理へ攻撃中を通知する，もしくは衝突判定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.
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.

保持する Tuple で，対応する人間以外の処理から，その状態を調べるために利用されていた．この Tuple はキャラクタと 1 対 1 としなければならないので，位置や体力が変化する時は，古い Tuple を削除し新しい値を持つ Tuple を書き込むようにすることで，更新処理を行なっていた．

「位置情報」Tuple や「誰」Tuple はキャラクタを生成する際に必要なパラメータを，生成処理に渡すために使用される Tuple である．生成すれば必要無くなるので，生成処理中に「取る」で削除されるようになっていた．

Tuple Space はゲームキャラクタ間の通信手段として様々な使われ方をしていたが，それらの記述からは次の問題点があることも判明した．

- バグの発生源となることが多い.
- Tuple Space では簡潔に記述できない通信がある.

S540 はサブルーチンの定義や呼び出しができず，異なるのキャラクタの

表 3.8: 実行時の主な Tuple のアクセス量

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
プレーヤーロボット	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 (約 230 秒間).
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

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 が必要で，Tuple の細かい操作記述が多く必要となっていた．そのため，Tuple の読み書きの間違いによるバグが多く発生しており，それを防ぐためにゲームデザイナー 3 人の間では Tuple の引数型や引数の数を識別子ごとに完全に決めて，同じ識別子で異なる型や数の引数を取る Tuple は使わないようにしていた.

また，Tuple Space の機能だけでは簡潔に記述するのが難しい記述も見られた．例えば，**表 3.7** の「状態」Tuple は人間の状態を表す Tuple であるが，最新の人間の状態を常に反映し続けなければならず，フレーム毎に Tuple の削除と書き込みの 2 つの処理が必要で，冗長な記述の要因となっていた.

記述が簡潔にならない他の例として，ある処理を特定のキャラクタ全てに対して行うような場合も挙げられる.「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 Tuple で，引数には存在する敵キャラクタへの参照を取る．この Tuple を利用して「全敵キャラクタについての処理」をするために次のように記述されたプログラムが 2 つ以上あると，先に実行されたプログラムにより敵ロボ Tuple が Tuple Space 上から削除されてしまい，後に実行されるプログラムでこの敵ロボ Tuple を使用できない問題がおこる [RW98].

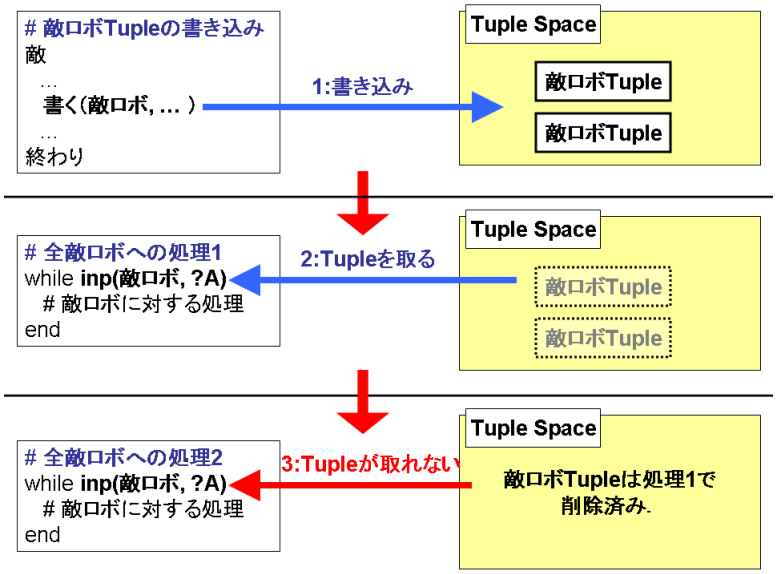

図 3.11: 敵ロボ Tuple を in する処理の Tuple の奪い合い

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

図 3.11 は，これを図示したものである．書きこまれた「敵ロボ」Tuple は，「全敵ロボへの処理 1」により削除されてしまい,「全敵ロボへの処理 2」の時点では存在せず，処理 2 は正しく実行されない．

全ての敵キャラクタについての処理を正しく行うためには，次のように全ての人間の中から「敵ロボ」の引数に一致するものだけを選択するように記述し，Tuple の削除をしないようにしなければならなかった．

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

これら「状態」や「敵ロボ」Tuple はどのキャラクタのものかを識別するために，関連しているキャラクタを引数として持つ．特定のキャラクタに関連付けられる Tuple は，他にも，プレーヤーキャラクタ判別用の「プレーヤーロボット」やキャラクタへアクションの指示をする「コマンド」がある．これらの Tuple の読み込みや書き込みは必ずキャラクタを引数として伴い，記述を冗長にする要因となる．

その他の記述の傾向

その他の記述の特徴として次のものがあった.

- ほとんど日本語文字により記述されていた.
- キャラクタ内部のデータの多くがキャラクタ内部の状態間通信のために使用されていた.
- 人間の動作処理の数値的なものは Excel 上で記述され，そのデータを読み込んで人間の処理を行うプログラムとなっていた.

ほぼ全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，概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者と半角を使う者がいたが，1 人で両方使っていることはなかった.

キャラクタ固有のデータ定義は当初，体力や攻撃力等のキャラクタ固有のデータを保持を目的としていたが，実際に使われていたものの多くは状態間でパラメータを受け渡しするためのものであった.

ゲームシーン中に登場する個々の「人間」は基本的な動作ロジックは同じだが，初期体力値や攻撃力等のパラメータや，グラフィクスデザインは異なる．類似したキャラクタすべてを異なるキャラクタとして記述するのは，プログラムを冗長にし管理も難しくする．そこで，人間の種類に応じたデータを表形式のデータとし，データ入力に Excel を使用するという開発スキームが作られた．この開発スキームのためには Excel の表形式データを読み込むライブラリが必要だったが，このライブラリの要求もゲームデザイナー側から出され，Excel のデータの処理は S540 で記述された.

3.5.6　使用後のインタビュー

S540 の有効性の評価をさらに進めるため，開発終了後，使用したゲームデザイナーにインタビューを行った.

インタビュー形式

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，ゲームシステム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ー 1 人の 4 人である．主な意見を**表 3.9** に挙げる.

言語の理解しやすさ

得られた意見から言語の理解しやすさについて次のことが読み取れる.

表 3.9: 主な意見

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	プログラマー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良いのかがわからない.
	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づらい.
	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.

- 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
- 自律した処理が通信しあうという記述モデルは自然に受け入れられた

「IF や GOTO があって BASIC のようだった」という意見から，プログラミング経験のあった 2 人にとって，S540 が既知のプログラミング言語と類似していて，基本的な部分での記述が難しいものではなかったということがわかる．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．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あり，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

S540 記述ツールを導入することによる開発進行への影響については次のことがわかる.

- トライアンドエラーが頻繁に行われた
- 意図する実験すべてができたわけではなかった
- 記述ツールの使用により開発形態がどのように変化するのかイメージできていなかった

プログラマーに頼まずにゲームシステムの実験ができた，という意見から，これまで彼らが経験してきたビデオゲーム開発と比較して，ゲームシステムの実験の範囲が広くなったと感じていることがわかる．また，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もわかる．

しかし，実験のための変更がゲームシステムプログラム全体に影響を及ぼす場合は，すぐに実験というわけにいかなかった，という意見や，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全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ーによる 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，ゲームシステム全てをスクリプト言語で記述する開発形態が，開発者にとって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

並行処理に関しては得られた意見から次の問題点があることがわかった．

- バグの修正が難しい．
- スレッドへの処理の割り当ての設計が難しい．
- スレッドの処理順序を指定したい．

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一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，キャラクタが存在するかどうかを保持する「敵ロボ」Tuple を用意し，適切にキャラクタが消される記述をしていた．

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上のイベントや通信に対して，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，処理の流れをもっと厳密にコントロールしたいという理由による．この問題は，次節で述べる，スレッドの処理順序により，Tuple Space を用いた通信が遅延することとも関連している．

Tuple Space について

Tuple Space はゲームシステムプログラム中で頻繁に使用されており，要望や問題指摘もあった．

- Tuple の上書き機能が欲しい
- ブロードキャスト機能が欲しい
- 通信遅延が発生する．遅延を回避する方法もない．

Tuple の上書き機能が欲しいという意見は，グローバル変数のような使い方をする Tuple を更新する場合，必ず一度 `in` により古い値を持つ Tuple を削除して `out` で新しい値の Tuple を書き込ま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．前節で述べたキャラクタの状態を表す「状態」Tuple もキャラクタの状態を反映するためには毎フレーム同様の操作をしなければならない．これらの操作は，状態を表す Tuple を上書きする機能か，書き込み時に対応する古い Tuple を自動で削除するような機能があれば冗長さを排除でき，より簡潔に記述できたと考えられる．

個々の通信ごとに `in` や `out` による書きすぎや消し忘れ等の，間違った Tuple の操作が無いように注意して記述することは，プログラミングに精通していないゲームデザイナーにとって簡単ではない作業であり，Tuple Space が簡潔でわかりやすい反面，キャラクタ間の通信の記述のための機能が不足していたと考えられる．

ブロードキャスト機能は，ある処理で `in` した Tuple を他の処理でも必要とする場合，後の処理は該当 Tuple が削除されてしまい `in` することができない [RW98] という理由からあれば便利という意見として上げられた．

並行処理の処理順序によっては，`out` した次のフレームでないと `in` できない場合がある．この通信遅延の指摘は，通信の順序が定義されておらず，それがゲームバランス上問題になったことから出された意見である．通信の順序や優先順位を記述する機能は，同時衝突や，同時に適用できるルールがある場合の解決方法として必要で，ゲームバランス上致命的になる可能性もあるため，改善が必要な部分である．

3.6 他の開発プロジェクトとの比較

S540 記述ツールを導入することによる有効性を評価する方法として，他のビデオゲーム開発プロジェクトとの比較が考えられる．

しかし，開発プロジェクトの期間や人数についての詳細なデータは一般には公開されることが少なく，同規模のデータを見つけることはできなかった．また，本章で説明した S540 記述ツールを用いたビデオゲーム開発はプロトタイプを目標にしており，完成した製品との比較は難しい．このプロトタイプ開発を行った開発会社にはビデオゲーム開発の経験のある社員が多いが，会

表 3.10: 本章のプロジェクトと他のプロジェクトの人数の比較

	4ヶ月		完成時	
プロジェクト	デザイナー	プログラマー	デザイナー	プログラマー
本章の プロトタイプ開発	3	2		
Prj-1([新宅 03])	2	9	8	19
Prj-5([新宅 03])	2	5	2	7

社としては新しく，過去の開発データが無いことから，社内における比較もできなかった．

文献 [新宅 03] には，ビデオゲーム開発プロジェクトの概略が数例報告されており，このうち 2 例については，時系列で開発規模の変化や進捗状況についてのデータも報告されている．この 2 例について，制作作業が始まって本章のプロトタイププロジェクトと同程度の 4ヶ月程度経過した時点でのゲームデザイナーとプログラマーの人数を，本章のプロジェクトと合わせて**表 3.10** に挙げる[7]．

この 2 例は完成に 1 年から数年かかっており，ゲーム内容については不明で，開発の進行具合についても概略しかわからないので，比較対象として必ずしも適切とは言えない．しかし，4ヶ月の時点で，Prj-1 はハードの検証や表現のテストが終わった段階であり，Prj- 5 は前作となるゲームプログラムの検証等を終え，新しいプログラムが動き始めた時期であるということから，プロトタイプでトライアンドエラーを繰り返した本章の開発プロジェクトと類似しており，比較は無意味ではない．

この比較から次のことを確認できる．

- 本章のプロトタイププロジェクトは，ゲームデザイナーの人数よりもプログラマーの人数が比較的少ない．
- 他のプロジェクトは，4ヶ月の時点で本章のプロトタイププロジェクトよりもプログラマーの人数が多い．

本章のプロジェクトにおいて，ゲームシステムはゲームデザイナーによって実装されており，プログラマーはゲームシステムに必要な機能をライブラリとして提供する作業に従事していたことから，ゲームデザイナーの方がプログラマーよりも多く必要とされた．一方，**表 3.10** に挙げた Prj-1 や Prj-5 では，その規模や人数から，ゲームシステムの実装をプログラマーが行なっており，そのため，本章のプロジェクトと比較してゲームデザイナーよりもプログラマーが多くなったと予想される．

また，他のプロジェクトのプログラマーの人数は，本章のプロジェクトよりも多い．Prj-1 と Prj-5 は共に検証やテストがプログラマーによって行われていることから，この時期にプログラマーが多く必要だったのは，その検証やテストの作業のためだったと考えられる．この点において，本章のプロ

[7] [新宅 03] ではゲームデザイナーのことを日本のゲーム業界用語である「企画」としている．

ジェクトでは検証やテストの多くをゲームデザイナーが進めたため，プログラマーの作業が必要となることは少なかった．

これまでに述べて来たゲーム開発にはトライアンドエラーが不可欠であるということと，ソフトウェア開発においては同じ作業量であれば少人数の方が効率がよいという一般に認められている知見 [FPB02] をともに考慮すると，本章のプロジェクトのような、より少ない人数でゲームデザイナーが中心となってトライアンドエラーを行うことを可能にする S540 記述ツールの使用は有効であるものと考える．

3.7 評価

開発で作られたプログラムの調査とインタビュー，及び他のプロジェクトとの比較から S540 記述ツールについて次の評価ができる．

- ゲームデザイナーにとって S540 は難しいものではなかった．
- トライアンドエラーを容易にする効果があった．
- ゲームデザイナーがプログラマーの協力無しに開発を進めることができた．

作られたプログラムやインタビューを通して，特定の言語機能の難しさが原因で作業が滞ることは見受けられなかった．およそ 4ヶ月の間に全ての言語機能が十分に使用されており，S540 は，ゲームデザイナー自身が記述を行うプログラミング言語として十分に利用可能なものであった．

また，3.5.6 節の使用後のインタビューから，ゲームデザイナーらは，それまでに体験してきた開発プロジェクトと比較して，より実験や検証をできるようになったと感じていた．さらに，3.6 節での他の開発プロジェクトとの比較から，本開発プロジェクトは，比較的少人数で開発が進められた．これらのことから，S540 記述ツールの導入により，ゲームデザイナーだけで，多くの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でき，トライアンドエラーを容易にする効果があったと評価できる．

ゲームシステムに関するテストや検証の多くがゲームデザイナー自身により行われ，プログラマーはゲームシステムに関与していなかった．このことから，S540 記述ツールを使った開発において，ゲームデザイナーは，プログラマーの協力無しにゲームシステムの制作を進められたことがわかる．さらに，3.5.5 節で述べたように，Excel を使った開発スキームの構築に対して，ゲームデザイナーが積極的に関わっていることから，ゲームシステム全体を実装できることで，開発プロジェクトの進行について，より積極的に参加を促す効果もあった．

一方で，次の問題点があることも明らかになった．

- 言語の機能が十分でない．

- 開発環境が十分整備されていない．

言語の機能が十分でないために，いくつかのゲーム要素の記述には冗長な方法を使わなければならなかったり，実装が難しくなる場合があることが判明した．手続き機能が無いので，プログラム中に同じ処理を繰り返し書く必要があり，プログラムの規模が大きくなりやすいことも確認できた．

特に Tuple Space は理解しやすく様々な通信の記述が可能ではあるが，次の問題があることも明らかになった．

- Tuple 操作の記述が冗長になやすく，簡潔さの点で問題がある．
- Tuple の処理順序やタイミングを明示的に記述できない．
- 多数のキャラクタ間の通信を記述する場合，Tuple の読み書きが煩雑になる．

Tuple 操作の記述の冗長さの問題は，3.5.5 節で述べた，キャラクタの状態を保持する Tuple は，情報の更新のために毎フレーム古い Tuple の削除と新しい Tuple の書き込みが必要であったことや，使用後のインタビューにおける上書き機能の要求から明らかになった．

Tuple の処理順序やタイミングを明示的に記述できない問題は，並行処理についてのインタビューで,「スレッドの処理順序を指定したい」「通信遅延が発生する．遅延を回避する方法もない」という意見から明らかになった．スレッドの処理順序によっては，Tuple を `out` した次のフレームでないと `in` できない場合があり，これを原因としたゲーム進行上のイベントや通信に対するキャラクタ反応の遅れは，ゲームバランス上致命的となることがあった．

さらに，多数のキャラクタ間の通信を記述する際に Tuple の読み書きが煩雑になる問題は，3.5.5 節で説明した「敵ロボ」Tuple のように，複数の異なる処理から同一の Tuple を削除せずに処理する記述が簡潔にできないことが原因となっている．この問題は，インタビューで得られた，複数のキャラクタにブロードキャスト機能が欲しいという意見からもわかる．

デバッグが難しいという意見から開発環境が十分に整備されていないことも問題点として挙げられる．特に並行処理と通信に関するバグは発見することが難しい場合があり，デバッグを容易にするためのツールや環境が必要である．

3.8　まとめ

本章では，新しく開発したプログラミング言語 S540 とその記述ツールについて説明した．

S540 は，1.9 節で述べた,「キャラクタの状態遷移処理」「時間に沿った処理」「並行処理」「キャラクタ間の通信」の 4 つの要件を満たし，プログラミングに精通していない，ゲームデザイナーが直接利用しやすい設計とした．

4 つの要件のうち,「キャラクタの状態遷移処理」と「時間に沿った処理」は，処理を管理するための変数や条件分岐を必要とせず，処理内容を簡潔に明示できる記述方法とした.「並行処理」は，生成したキャラクタが自律的に動作し，キャラクタの記述内では====という区切り記号を用いてわかりやすく簡単に複数の並行処理を記述できるようにした.「キャラクタ間の通信」の記述機能としては，制御の流れが複雑にならず，操作が Tuple の読み書きだけというシンプルで理解が容易な通信モデルである Tuple Space を採用した.

S540 はその他に，開発中に使用する言葉を直接ゲームシステムプログラム中で使用できるようにするための日本語文字を用いた記述機能や，ゲームシステム全体を記述可能にするための構造，実行処理時間を最小限にするためにゲームシステムプログラムは C 言語プログラムへ変換されてアプリケーションに組み込まれるという特徴を持つ.

この設計に従い開発した S540 記述ツールを，実際のビデオゲーム開発の現場でプロトタイプゲームの開発に使用した．プロトタイプ開発プロジェクトでは，ゲームシステムは全てゲームデザイナーにより記述されたことと，書かれたゲームシステムプログラムや使用後のインタビューから，S540 は，ゲームデザイナーがプログラマーの協力が無くとも，ゲームシステムを記述することを可能にする機能を持つことを確認できた．また，S540 記述ツールの導入で，利用者らがそれまでに経験した開発プロジェクトよりもトライアンドエラーが容易になったことがわかり，本研究の目的がある程度達成されたことを確認できた.

また，この結果から，1 章で議論した 4 要件を満たすことで，プログラミング言語はゲームシステムを開発する際のトライアンドエラーが容易になるということも，間接的であるが確認することができたと考える.

しかし，S540 には問題点も確認された．特に，キャラクタ間の通信の記述方法として導入した Tuple Space は，シンプルなモデルであることから理解が容易で，様々な通信の記述が可能だが，キャラクタ間の通信の記述は冗長となったり，煩雑な操作を必要とする場合があることがわかった．また，通信の処理順序やタイミングを明示的に記述できず，ゲームバランス上問題になる場合があることもわかった.

キャラクタ間の通信は，ビデオゲームにおいて特に複雑になる部分である．本章の評価から，ゲームシステムを記述するプログラミング言語には，Tuple Space に代わる，よりビデオゲームに適した通信モデルが必要であることがわかった．次章では，そのような新しい通信モデルについて述べる.

第4章　キャラクタ間通信の記述性の改善

4.1　はじめに

S540 はアクションゲームのゲームシステムを十分に記述できる機能を持つことが確認できた．キャラクタ間の通信を記述するために導入した Tuple Space もシンプルで理解しやすく，様々な通信の記述が可能である．しかし，Tuple Space には，記述が冗長になりやすい，通信の処理順序やタイミングを明示できない，煩雑な操作を必要とする場合がある，という問題があることも明らかになった．

これらの問題を解決するために，新しい通信モデル Join Token[NK11a, NK11b] を考案した．Join Token は，Tuple Space に join calculus の特徴を融合した通信モデルで，Tuple Space を利用した場合に必要な細かい操作の記述を軽減し，通信の処理順序を明示できる等の機能を持つ．

この Join Token を評価するために，オブジェクト指向プログラミング言語 Mogemoge を開発し，Mogemoge 上に Join Token の構文設計及び実装を行った．また，言語 Mogemoge を使用して複数のゲームアプリケーションの制作を行い，その評価を行った．さらに，制作したゲームと同一のものを，Join Token を使用しないプログラミング言語で制作し，両者の比較による評価も行った．

以下，本章では，4.2 節で前章で明らかになった Tuple Space の問題点について再度議論し，4.3 節でその問題点を克服する新しい通信モデル Join Token について述べる．4.4 節では Join Token を実装したプログラミング言語 Mogemoge について説明し，4.5 節で Mogemoge を使った Join Token の評価について説明する．最後の 4.6 節では，本章を総括する．

Join Token を実装したプログラミング言語 Mogemoge の構文定義は，本論文の末尾 B.1 節 に付録として掲載している．また，本章の研究に際して制作したゲームの全ソースコードも B.2 節，B.3 節，B.4 節，B.5 節に掲載している．

4.2　Tuple Space の問題と解決方法

4.2.1　Tuple Space の問題点

Tuple Space については，3.7 節で次の問題点が明らかになった．

- Tuple 操作の記述が冗長になやすく，簡潔さの点で問題がある．
- Tuple の処理順序やタイミングを明示的に記述できない．
- 多数のキャラクタ間の通信を記述する場合，Tuple の読み書きが煩雑になる．

Tuple Space では Tuple の一意性は考慮されず，全く同じ引数の Tuple であっても，異なる Tuple として扱われる．そのため，グローバル変数やキャラクタの状態を表す目的で Tuple を用いる場合，2 つ以上の同じ意味の Tuple が存在しないよう，値更新時は，古い Tuple を削除してから新しい Tuple を書き込むようにしなければならない．書き込みと読み込みを常にセットで記述することは，記述が冗長になり簡潔さの点で問題がある．バグの発生原因にもなりやすい．

あるキャラクタ間の通信を Tuple Space で実装した場合，受信側の Tuple を読むタイミングは，並行処理の実行順序に依存する．S540 には処理順序を明示的に記述する方法が無いので，書き込み直後すぐに処理してほしい Tuple や，ある処理の前に読み込んでほしい Tuple があっても，そのとおりに動作するよう記述することが難しい．また，通信は複数の並行する処理間で行われるので，個々の並行処理の処理順序を指定できるだけでは，通信の処理順序を管理することができない．

多数のキャラクタに影響のある衝突処理やゲーム中のイベント処理の場合，Tuple を消さずに全ての Tuple について処理をする記述が難しい．そのため，何度も Tuple Space 上で Tuple のやりとりを行ったり，余計な繰り返し処理が必要となり，ブロードキャスト機能が欲しいという意見になっていた．

4.2.2　問題点の解決方法

前述した，キャラクタ間の通信を記述する上での Tuple Space の問題点を解決するためには，Tuple Space に次の機能が必要である．

- 名前や関連するキャラクタで識別が可能な Tuple
- 通信を個々のキャラクタの処理とは分け，処理順序を明示する機能
- ある複数の Tuple に対して一括して通信を記述できる機能

Tuple を名前で識別可能にすることで，Tuple の上書き機能を導入し，Tuple Space では冗長になりやすい記述を簡潔にすることができる．また，Tuple にキャラクタを関連付けられるようにすることで，3.5 節で述べたような，キャラクタの状態を Tuple で表現しやすくなる．

通信を明示的にキャラクタの処理から分けて記述し，処理順序を明示できれば，並行する処理の処理順序には関係なく，通信の処理順序を記述することができる．

多数のキャラクタ間の通信を簡潔に記述可能にするためには，ある特定の Tuple 群に対して一括した通信を記述できる機能が必要である．

2.6.1 節で述べた join calculus は，メッセージの送受信を行うチャネルを使って複数の並列動作するプロセス間の通信を表現するモデルである．このチャネルは複数のメッセージを同時に受信する join pattern という記法があり，複数のプロセス間の通信をわかりやすく記述できる．join calculus のプロセスをゲームキャラクタの処理と捉えれば，キャラクタ間の通信はチャネルで表現され，通信を個々のキャラクタの処理とは分けて記述可能になると考えられる．

しかし，join calculus のチャネルを用いた，通信時に新しいプロセスが起動するという仕組みは、ビデオゲームの構造と対比して理解することが難しい．join calculus ではプロセスの処理順序は定義されないので，チャネルの処理順序を定義することはできない．また、データとして表現されないメッセージが同時に複数やりとりされる仕組みは、プログラミングの経験があまりない人間にわかりやすいモデルではない．

そこで，Tuple Space と join calculus の特徴を併せ持ち，上記の 3 つの機能を持つ新しい通信モデル Join Token を考案した．次節では，この Join Token について説明する．

4.3 Join Token: Tuple Space と Join 機能の統合

4.3.1 Join Token の概要

Join Token[NK11a, NK11b] は，Tuple Space と join calculus の特徴を併せ持った新しい通信モデルである．

Join Token は次の要素から構成される (**図 4.1**).

- 名前と引数列で構成されるトークン
- トークンを溜めるトークンプール
- トークンプール内のトークンを基に起動されるハンドラ

Join Token では通信に，識別のための名前と引数列を組にした「トークン」を使用する．トークンは，通信に関係するキャラクタが必要に応じて生成する．生成時，トークンはトークンを生成したキャラクタと関連付けられ，どのキャラクタが生成したトークンかを識別できる．

「トークンプール」は，Tuple Space と同様に，どの処理からも自由にアクセス可能なグローバルな場所である．全てのトークンは，生成されるとトークンプールに入れられる．

Join Token におけるキャラクタ間の通信は「ハンドラ」で記述する．ハンドラは，以下のように，トークンパターン，起動条件及び処理本体で定義される．

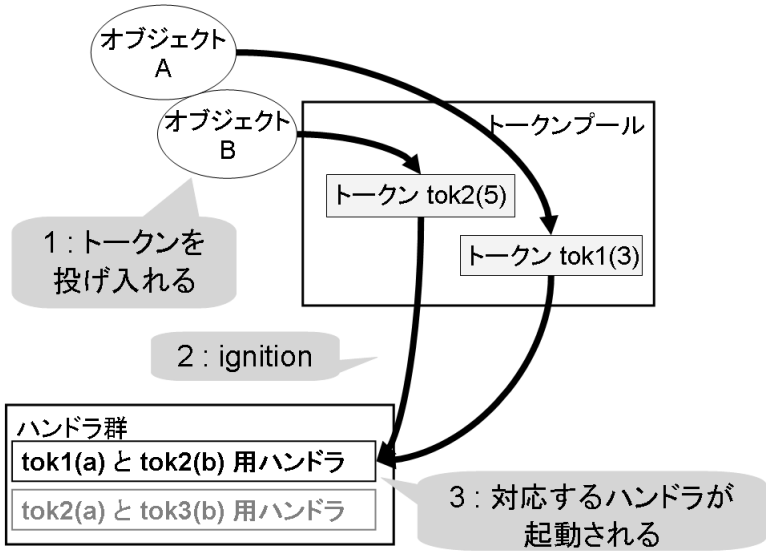

図 4.1: Join Token の概念図

- トークンパターンは，処理に必要なトークンをトークンプール内から識別するためのものである．トークンのマッチングはトークンの名前で行う．トークンパターンは複数のトークン群にマッチさせることができる．
- 起動条件はハンドラの起動条件を表す論理式である．トークンパターンにマッチするトークン群がトークンプールにあっても，起動条件を満たさない場合，ハンドラは起動されない．
- 処理本体は，トークンパターンと起動条件が満たされたときに実行されるコードである．処理本体では，トークンパターンによりマッチしたトークンに関連付けられたオブジェクトや引数列の値を参照することができる．

Join Token における通信は (**図 4.1**) は次の順に行われる．

- キャラクタがトークンをトークンプールに入れる．
- 発火処理を実行する．
- 発火処理が全トークンの組み合わせについてマッチするハンドラを起動する．

通信に関係するキャラクタは，適当なトークンを生成しトークンプールに投入する．例えば，プレーヤーキャラクタが関係する通信を行うハンドラが，“player” という名前のトークンを処理するよう記述されているとする．この

とき，プレーヤーキャラクタの生成時に，“player” という名前のトークンをトークンプールに投入すれば，生成したキャラクタをそのハンドラの処理に適用することができる．

キャラクタ間の通信は，個々のキャラクタではなくハンドラにより処理される．ハンドラは発火処理という命令により明示的に起動される．発火処理はキャラクタの処理とは独立しているので，通信のタイミングをキャラクタ処理とは別に記述することができる．

発火処理が実行されると，全ての定義されているハンドラについて，トークンプール上の全てのトークンのマッチングが行われ，マッチするトークンが見つかり起動条件も満たすハンドラがあると，そのハンドラの処理本体が実行される．2 つ以上のトークンの組み合わせにマッチするハンドラの場合，マッチングはトークンプール上のトークンの組み合わせ全てについて行われる．起動されるハンドラの処理本体には，マッチしたトークンの引数や関連付けられたキャラクタを参照した処理を記述する．全てのトークンとハンドラのマッチングが終わると，発火処理は終了する．

4.3.2 Join Token の処理手順

Join Token では，キャラクタ間の通信の処理順序を明示的に記述可能とするために，各機能の処理内容を詳細に定義している．

同一キャラクタが同一名のトークンを生成してトークンプールに投入した場合，先にあった古いトークンが新しいトークンに置き換えられる．これにより，Tuple Space では冗長となりやすかった上書きの機能を実現している．

ハンドラは処理本体実行時に，トークンをトークンプールに残しておくか削除するかを指定することができる．残しておくことで，キャラクタの状態を表すようなトークンを他のハンドラでも利用することができ，削除することで，3.5.5 節の人間キャラクタに対する「コマンド」のような，1 回限りの処理を記述することもできる．

ハンドラの起動条件が成立しない場合，ハンドラの処理本体は実行されず，トークンの操作も行われない．従って，ハンドラがトークンを削除するようになっていても，起動条件が成立しなければトークンは削除されない．

発火処理時，ハンドラは記述順序に従って走査される．トークンプール上のトークンにマッチするハンドラが複数ある場合は，記述の上でより先に記述されているハンドラから走査される．これにより，ハンドラの処理順序を記述者が明示することができる．

同様にトークンについても処理順序が定められている．トークンは，ハンドラとのマッチング処理時，トークンプールに投入された順序で走査される．1 つのハンドラが複数のトークンの組み合わせにマッチする場合，より先に投入されているトークンを含む組み合わせからハンドラとのマッチングを行う．

ある発火処理時に，ハンドラの処理本体で新たにトークンが生成され，トークンプールに投入されても，それまでに走査されたハンドラが再度走査対象

になることは無い．この仕組みは，マッチング処理が無限に続くことと，処理順位の高いハンドラが低いハンドラよりも後に実行されるのを防ぐ．

一方，ハンドラの処理本体で生成されたトークンは，より処理順位の低い未走査のハンドラには適用される．これにより，処理順位の高いハンドラから低いハンドラへ，トークンを使った連鎖的な処理の記述が可能になる．

4.3.3　Join Token の特徴

Join Token は，Tuple Space と join calculus モデルを融合した通信モデルであり，ゲームシステムにおけるキャラクタ間の通信の記述に適した次の特徴を持つ．

- Tuple Space で冗長になりやすかった記述を簡潔にできる．
- 通信の処理順序や，通信のタイミングを明示的に記述できる．
- ハンドラのトークンパターンと発火時のマッチング処理により，多数のキャラクタ間の通信を一括して記述できる．
- キャラクタの種別や状態に依存した通信が記述できる．
- ゲームプログラムの構造に適合している．
- 並行処理やオブジェクト指向の通信モデルとして適している．

Tuple Space では，Tuple を一意に識別する仕組みがないので，Tuple をグローバル変数やキャラクタの状態を表す目的で使用するには，古い Tuple の削除と新しい書き込みを組にして記述しなければならない．Join Token では，トークンの名前とトークンを生成したオブジェクトの組に基づいてトークンの識別が可能であり，同名同一生成元のトークン書き込みは古いトークンを削除できるようにすることでこの問題を解決している．

また，Tuple Space では Tuple の読み書き処理が並行処理の処理順序に依存するため，通信の順序を明示できないが，Join Token ではハンドラの記述順序で処理順序を明示できる．そして，発火処理としてハンドラの起動タイミングを記述できるようにすることで，トークンの生成を含む個々のキャラクタの処理から通信についての記述を分離し，キャラクタの処理とは関係無く，通信のタイミングの明示が可能である．

さらに，Join Token では，ハンドラはマッチするトークンの組み合わせ全てに対して適用される．3.5.5 節の「敵ロボ」Tuple の例のように，Tuple Space では条件に一致する Tuple 全てについての処理を記述する場合，Tuple を削除せずに走査するのは簡単ではない．一方，Join Token ではトークンパターンにマッチするトークンの組み合わせ全てについてハンドラが起動されるので，複数のトークンのやその組み合わせについての処理を 1 つのハンドラで一括して記述できる．

Join Token が持つ Tuple Space には無い特徴として，キャラクタの種類や状態に依存した処理が記述しやすいことが挙げられる．トークンの名前をキャラクタの種類や状態に対応させることで，特定のキャラクタや特定の状態のみ処理するハンドラを記述することができる．

ビデオゲームアプリケーションは 1/10〜1/60 秒単位のフレーム処理を無限に繰り返すことでアニメーションの表現を実装する．ゲームプログラムはこの実装のために，次のような 1 フレーム分の「キャラクタ状態の更新」と「キャラクタの状態更新によるイベント等の処理」を無限に繰り返す実装とすることが多い．

```
while ( true )  # 無限ループ
  全キャラクタの更新処理
  更新に伴うイベント等の処理
end
```

無限ループ内の全キャラクタの更新処理ではキャラクタのトークン生成を行い，更新に伴うイベント等の処理は発火処理として記述できる．このように，Join Token はゲームプログラムの構造に適した処理モデルとなっている．

Join Token は，キャラクタ間の通信を，並行処理によるトークンの投入と通信処理をするハンドラに分離している．このため，個々の並行処理の制御の流れが通信の記述により複雑になることがなく，並行処理間の通信を記述するのに適している．また，トークンは必ずオブジェクトに関連付けられることから，オブジェクトが関係する通信を表現しやすく，キャラクタ間の通信を処理するモデルとしても有効である．

4.4 Join Token 機構を持つプログラミング言語 Mogemoge

4.4.1 Mogemoge 言語の概要

Join Token の有効性を評価するために，新しく開発したプログラミング言語 Mogemoge において，構文も含めた Join Token の設計と実装を行った．本節では，Mogemoge と実装した Join Token 機構について説明する．

Mogemoge は JavaScript[Fla07]，Self[US87]，Dolittle[兼宗 01] に似たプロトタイプ型のオブジェクト指向言語である．

Mogemoge には，S540 の様な状態遷移や並行処理を記述する機能は無く，他の汎用プログラミング言語や S540 と比較してシンプルな構文となっている．これは，Join Token 自体の評価を行いたいことと，Join Token の構文や実装を簡潔にするための二つの理由からである．

文

Mogemoge プログラムは，文の集まりとして記述する．Mogemoge の文には次の 4 種がある．

- 代入文
- 制御文
- 式
- Join Token 処理文

代入文は，変数に値を設定する文である．次のように=を使い記述する．

```
a = 5; # 変数 a に値 5 を代入
```

文は; 記号で区切られる．同じ変数に何度も代入することができる．変数は代入文によって定義され，代入文が無いまま参照すると誤りとして報告される．

制御文は，他のプログラミング言語でも用意されている条件分岐や繰り返し処理の制御をするもので，`if` や `while` 等がある．次は，変数 `n` を 1 から 10 まで変化させながら繰り返し `n` の値を表示する `while` 文の例である．

```
n = 1;
while (n <= 10) {
  print "n=" + n;
  n = n + 1;
}
```

式は，単体で文となる．式には，+や-等の四則演算の他に，比較演算子やメソッド呼び出し等がある．

Join Token に関係する文については，別節で詳細を説明する．

メソッド

メソッドは文をまとめた手続きである．メソッドは呼び出す際に引数を渡したり，メソッドの処理終了時に呼び出し側に戻り値として処理の結果を返すことができる．

次は簡単なメソッドの定義と呼び出しの例である．

```
foo = method(n) { result n * 2; };
a = foo(5);  # a = 10
```

Mogemoge においてはメソッドは整数や文字列等と同様に値である．メソッドは次の形式で記述する．

```
method(仮引数列) { 文の列 }
```

この記述により生成したメソッドを変数に代入することで，メソッドを変数名で識別することができる．上記の例では，変数 `foo` へ引数を 1 つとるメソッドを代入し，この記述以降 `foo` という名前で呼び出し可能にしている．

メソッドを表す式 (上記の例では `foo`) に続いて，`()` を記述することでメソッドを呼び出すことができる．

手続き呼び出しが利用できる他のプログラミング言語と同様，Mogemoge のメソッドも呼び出し時に引数を渡すことができる．また，メソッドは `result`

文でメソッドの返す値を指定することができる．result 文は，C や Java 言語等の return 文と異なり，メソッドの処理を中断しない．

メソッドが値であることから，次の例のような，メソッドを返すメソッドを定義することもできる．my は局所変数を導入する修飾語である．

```
foo = method() {
  my x = 1;
  result method(d) { result x; x = x + d; }
};
f = foo();  # f = メソッド
b = f(5);   # b = 6 (1+5)
c = f(3);   # b = 9 (6+3)
```

オブジェクト

Mogemoge では他のオブジェクト指向言語と同様に，変数やメソッドをまとめたオブジェクトを記述することができる．Mogemoge はプロトタイプ型言語でクラスはなく，オブジェクトを定義として直接記述し，オブジェクトのコピーを作ることで，同種のオブジェクトの生成を行う．

次は，Mogemoge における銀行口座オブジェクトのサンプルコードである．

```
# Account 変数にオブジェクト代入
Account = object {
  v = 0;
  deposit = method(n) { v = v + n; };
  withdraw = method(n) {
    v = v - n;
    if (v < 0) { v = 0; }
  };
  get = method() { result v; };
};

a = new Account; # 新しい口座オブジェクト生成
a.deposit(100);
a.withdraw(50);
print "outstanding : " + a.get();
```

オブジェクトの生成は object {} を使う．object {} の中では，変数への代入を使い，オブジェクトに属する変数やメソッドの記述をする．

Account はオブジェクトが代入された変数である．new 演算子はオペランドのオブジェクトのコピーを返す．上記の変数 a には Account オブジェクトのコピーが代入される．depoosit や withdraw 等のメソッドの定義も，オブジェクトに属する変数への代入として表されている．

メソッドの呼び出しは Java のようなドット記法により表す．a.deposit(100); は，変数 a の保持するオブジェクトの deposit メソッドを，100 を引数として呼び出している．

全ての文は必ず何らかのオブジェクトに属する．プログラム全体は暗黙のうちに 1 つの object {} で囲まれており，オブジェクトの記述の外側で定義された変数やメソッドは，そのプログラム全体を囲っているオブジェクトに

属しているとされる．上記の例で，Account オブジェクトの外側にある変数 a は，その全体を囲うオブジェクトに属する変数として生成される．

オブジェクトの合成

Mogemoge には，他のオブジェクト指向言語にある継承や mix-in 等の機能はないが，代わりに 2 つのオブジェクトを合成して，両方の性質を持つ新しいオブジェクトを作る機能がある．次はその合成の例である．

```
A = object {
  foo = method { print "A.foo"; };
};
B = object {
  bar = method { print "B.bar"; };
};

C = A + B; # A と B を合成
C.foo();   # "A.foo"と表示
C.bar();   # "B.bar"と表示
```

この例では A という変数に代入されたオブジェクトと，B という変数に代入されたオブジェクトを，+演算子で合成し，C 変数に代入している．

C 変数は，両方の性質を併せ持つ新しいオブジェクトを保持し，A のメソッドも B のメソッドも呼び出すことができる．

どちらにも同じ名前のメソッドや変数がある場合は，+演算子の右側のオブジェクトのものを持つ．

4.4.2　Mogemoge 言語における Join Token 記述

本節では，Mogemoge 上での Join Token の記述について説明する．次のコードは 3 つのトークンをトークンプールに投げ入れた後に発火命令 (ignition) を実行する．ignition によって tok1 と tok2 がハンドラによって処理される．

```
join r1.tok1(a, b) r2.tok2(c, d) {
  print "a + c = " + (a + c);
  print "b + d = " + (b + d);
};
throw tok1(1, 2);
throw tok2(30, 40);
throw tok3("hello");
ignition;
```

throw 文はトークンをトークンプールに投入する．上記の例では，最初に名前 tok1 で引数 (1,2) を持つトークンがトークンプールに投入される．次に名前 tok2 で引数 (30,40) を持つトークンが投入され，最後に名前 tok3 で引数 ("hello") を持つトークンが投入される．

ignition 文は発火命令である．ignition 文を実行すると，全てのハンドラがトークンプール内の全てのトークンの組み合わせに対して検査され，一

致したハンドラが起動される．このとき一致したトークン列が実引数としてハンドラに渡される．上記の例では，tok1(1, 2) と tok2(30, 40) が，後述するハンドラのトークンパターンと一致するので，この二つのトークンを実引数としてハンドラが起動される．

join 文はハンドラを定義する．join 文のトークンパターン定義 r1.tok1(a, b) は，名前 tok1 で二つの引数を持つトークンと一致するパターンを意味する．a，b には一致したトークンが持つ引数が代入される．r1 にはこのトークンパターンに一致したトークンを投げ入れたオブジェクトが代入される．

tok1 と tok2 はハンドラに渡されてトークンプールから取り除かれるが，tok3 はハンドラによって処理されないので，ignition 文実行後もトークンプールに残る．

ハンドラの起動条件は where 節で指定する．次は，2 つのトークンが持つ引数が同一の時のみ起動するハンドラの例である．

```
join r1.tok1(a) r2.tok2(b)
     where a == b { ... };
```

処理したトークンをトークンプールから取り除きたくない場合は，パターンに “*” を指定する．次は，起動されても tok2 をトークンプールから取り除かないハンドラの例である．

```
join r1.tok1(a) *r2.tok2(b) { ... }
```

トークンプールに残されたトークンは他のハンドラの検査時に再利用される．どのハンドラでも利用されないトークンは次の ignition までトークンプールに残る．

投げ入れたトークンを明示的に取り除くには dispose 文を用いる．次は，トークンプールに tok1 があればそれを取り除く例である．

```
dispose tok1;
```

dispose 文が取り除けるトークンは，dispose 文を実行したオブジェクトが投げ入れたものに限定される．

トークンの一意性はトークン名と投げ入れたオブジェクトで決定される．あるオブジェクトが同じ名前のトークンを 2 度 throw すると，先に throw されたトークンは後に throw されたトークンで上書きされる．次は，tok(1) が tok(2) によって上書きされる例である．

```
A = object {
  m = method() {
    throw tok(1);
    throw tok(2); # 前の tok(1) を上書き
  };
};
A.m();
```

しかし，次の tok(1) は tok(2) と投げ入れるオブジェクトが異なるので tok(2) で上書きされない．

```
A = object {
  m = method() { throw tok(1); };
};
B = object {
  m = method() { throw tok(2); };
};
A.m(); B.m();
```

トークンの存在を確認するには，exist 演算子を使用する．次のコードは，このコードを実行したオブジェクトが投げた tok が存在するかどうかを調べる．

```
if (exist tok) { ...  }
```

ゲームプログラムでキャラクタの状態をトークンの集合で表している場合，exist 演算子はキャラクタのメソッド内で自分の状態を確認するために利用できる．

4.4.3　Mogemoge と Join Token の実装

Mogemoge と Join Token は，次のように実装した．

- Mogemoge は，Java 言語と SableCC を使用して実装．
- Join Token は，Java 言語の標準的なデータ構造を用いて実装．

Mogemoge は S540 と同様に，Java 言語と構文解析プログラム生成系の SableCC を用いて実装している．S540 と違いインタプリター形式で，処理系は読み込んだソースプログラムを抽象構文木データ構造に変換し，その木構造中の枝や葉に当たるデータを一つ一つ確認しながらプログラムの実行を進める．

Mogemoge 上の Join Token は，Java 言語と Java 言語の標準的なデータ構造であるリストやハッシュ等を用いて実装している．ハンドラは，条件式や処理本体の表す抽象構文木を持つ Java オブジェクトとして実装され，トークンは同名のものをまとめて 1 つのリストにして保持し，トークンプールはそのリストを辞書データ構造で管理する．

多くのトークンが存在する場合，ハンドラとトークンのマッチング処理は，実行時の処理時間に影響を与える可能性がある．単純に全てのトークンの組み合わせ 1 つ 1 つに対して全ハンドラのマッチングを処理するように実装した場合，M 個のハンドラが定義されている上で，トークンプール上に N 個のトークンがあると，マッチング処理の計算量は $O(MN)$ となり，決して高速であるとは言えない．よって，キャラクタが増え，ハンドラやトークンが多くなった場合に処理速度が問題となる可能性はある．

その際は，現在の実装を，Rete アルゴリズム [For82] 等のより高速なパターンマッチングアルゴリズムに置き換えて問題を解消することが可能である．Rete では，処理を高速にするための方法の 1 つとして，事実 (Mogemoge のオブジェクトとその状態にあたる) が変化すると，変化した事実に対応するルール (Join Token のトークンパターンと `where` 節) のみを再評価するアルゴリズムを採用している．このアルゴリズムは，ハンドラの `where` 節が，ハンドラを含む外側の要素全て (例えばグローバル変数等) を参照可能な現在の Mogemoge の構造では実装が難しい．よって，Rete アルゴリズムを採用する場合は，`where` 節ではトークンパターン内の仮引数しか参照できないようにする等の修正が必要となる．

しかし，トークンのマッチングは名前のみに基づき，トークンプールは名前を基にした辞書データ構造で実装されていることから，トークンの名前による検索処理にかかる時間は僅かである．また，現在の 3D グラフィクス等の処理で扱う頂点データ数 (数十万頂点以上) と比較すると，ゲームシステムで必要とされるキャラクタ数は僅かであることから，Join Token の処理が，ゲームプログラム全体の処理時間に与える影響は小さいと考えられる．

4.5 Join Token の評価

4.5.1 評価方法

Join Token の評価をするために簡単なゲームの制作を行った．ゲームは 3 つ制作したが，いずれも多くのゲームが採用しているゲームルールを含むゲームシステムとなっている．

ゲーム制作と評価は，目的とするゲームのルールを箇条書き形式で全て列挙し，それらのルールを，個々のキャラクタ固有のものか，複数のキャラクタが関係するルールか，により分類することから始める．その後，Mogemoge でゲーム制作を行い，複数のキャラクタが関係するルールがそれぞれ Mogemoge でどのように記述されたかを検討する．

また，ゲームのうち 1 つを Join Token 機能の無いプログラミング言語 Ruby で実装し，Mogemoge で記述したプログラムとの比較検討を行った．

制作したゲームの全ソースコードは本論文の末尾の B.2 節，B.3 節，B.4 節，B.5 節に掲載している．

4.5.2 シューティングゲーム Shooting

図 4.2 は Join Token と Mogemoge を使用して制作した，シンプルなシューティングゲームの画面である [NK11a].

このプログラムは，船 (図の鏃形) を操作して他の船を倒していくゲームである．プレーヤーは船の一つをキーボードで操作することができる．プレー

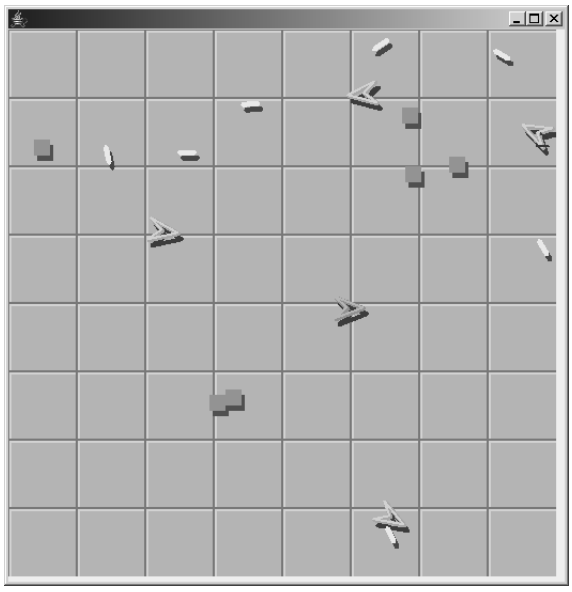

図 4.2: Shooting ゲームの画面

ヤーが操作するもの以外の船はプログラムによって操作される．プレーヤーはプログラムによって操作される全ての敵を撃破しなければならない．

以下，このゲームのルールをまとめる．

R1. 船はプレーヤーかプログラムによって操作される．

R2. 船はお互いに重なり合わない．

R3. 船はミサイルを撃つことができる．ミサイルは船にダメージを与えることができる．

R4. power food (図の四角形) を取ると，一定時間「無敵状態」になる．

R5. 無敵状態の船は，他の船に衝突することでダメージを与えることができる．

R6. 無敵状態の船は，ミサイルをダメージ無しに破壊することができる．

R7. 一定のダメージを受けると船は破壊される．

これらのゲームルールは，キャラクタ間の関係を表すルール (**R2**, **R3**, **R4**, **R5**, **R6**) と，キャラクタ固有のルール (**R1**, **R7**) に分類することができる．キャラクタ固有のルール (**R1**, **R7**) は，オブジェクトのメソッドで自然に記述することができるが，キャラクタ間の関係を表すルール (**R2**, **R3**, **R4**, **R5**, **R6**) は複数の種類のキャラクタやキャラクタの状態に依存するので記述が複雑になる．Join Token を用いることで，このキャラクタ間の関係についてのルールを簡潔に表現できる．

```
Ship = object {
  id = 0; x = 0;  y = 0;  dir = 0; timer = 0;
  init = method() { throw normal; }
  update = method() {
    if (exist unbeatable) {
      timer = timer - 1;
      if (timer < 0) {
        dispose unbeatable;
        throw normal;  # 船の状態の変更
      }
    }
    if (is_key_pressed(KEY_SPACE)) {
       m = new Missile;
       m.x = x;  m.y = y;  m.dir = dir;
       m.set_id(id);
    }
    # x,y,dir を更新する処理
  };
  make_unbeatable = method() {
    timer = 100;
    dispose normal;
    throw unbeatable;  # change ship's status
  };
  damage = method(d) { ダメージ処理 };
  is_collide = method(o) { 衝突検査処理 };
  # other methods ...
};
```

図 4.3: 船キャラクタのプログラム

図 4.3 は Mogemoge による船の実装の概観である．x,y,dir は船の位置や向きを表す変数である．ミサイルがどの船によって発射されたのかを識別するために，船は id に固有の整数 ID を保持し，その船から発射されたミサイルは同じ値の ID を持つ．

update は船の動きを処理するメソッドであり，アニメーションフレーム毎に呼び出される．is_collide は他のキャラクタとの衝突判定を行う．damage は船にダメージを与える処理を行う．船ごとに最初に一度だけ init が実行される．この中で，normal トークンがトークンプールに投げ入れられる．make_unbeatable メソッドが実行されると，normal トークンは取り除かれ，代わりに unbeatable トークンが投げ入れられる．

normal トークンと unbeatable トークンは船の状態を表している．update メソッドの中では，unbeatable トークンがトークンプールにあれば無敵状態とみなして，無敵時間 (timer 変数) の処理を行う．make_unbeatable メソッドは状態の切り替え処理である．

試作したゲームプログラムには Ship 以外に，Missile や PowerFood が定義されていて，Missile は最初に missile トークンを投げ入れ，PowerFood は最初に power トークンを投げ入れる．

図 4.3 がキャラクタ (船) 固有の記述であるのに対して，**図 4.4** は Join Token

```
# ルール R2
join *s1.normal() *s2.normal()
    where s1.is_collide(s2) {
  # s1 と s2 が重なり合わないようにする
};
# ルール R3
join *s.normal() m.missile(d)
    where s.is_collide(m) && s.id != m.id {
  s.damage(d);
  m.destroy();
};
# ルール R4
join *s.normal() p.power
    where s.is_collide(p) {
  s1.make_unbeatable();
  p.destroy();
};
# ルール R5
join *s1.unbeatable() *s2.normal()
    where s1.is_collide(s2) {
  s2.damage();
};
# ルール R6
join *s.unbeatable() m.missile(d)
    where s1.is_collide(s2) {
  m.destroy();
};
```

図 4.4: 複数のキャラクタが関わるゲームシステムに対応するハンドラ記述

を使用した複数のキャラクタに関係するルールについての記述である．

この記述の中で，複数のキャラクタに関係するルール (**R2**, **R3**, **R4**, **R5**, **R5**) は一対一対応で次のように表現されている．

- **R2** は，通常状態 (`normal`) 同士の船の衝突についてのルールである．ハンドラは二つの `normal` トークンを取ることで，通常状態の船キャラクタ間の処理であることを表し，衝突している場合にのみ実行される条件が指定されている．

  船の状態を表すトークンはこのハンドラの処理以外のハンドラにも必要なので，トークンプールに残される．

- **R3** は，通常状態の船とミサイルの衝突についてのルールである．ハンドラは `normal` トークンと `missile` トークンにマッチする．衝突時で且つ自分が発射していないミサイルという条件が指定されている．

  船はこの処理後も存在するので，`normal` トークンは取り除かれないが，ミサイルは消滅するので，`missile` トークンはトークンプールから取り除かれる．

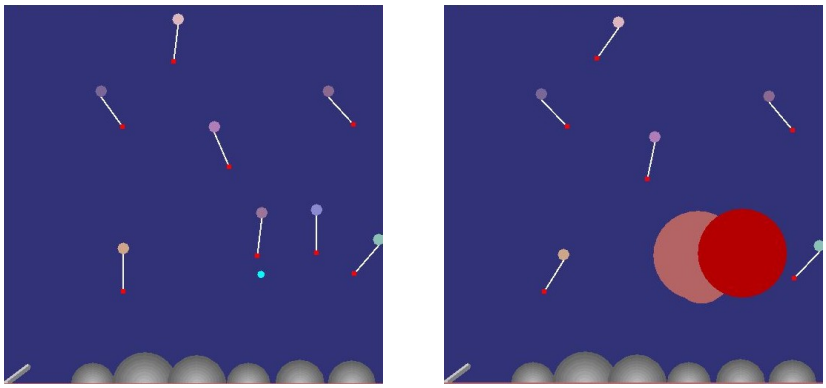

図 4.5: Balloon ゲームの画面

- **R4** は，通常状態の船キャラクタがパワーフードを取ったときのルールである．ハンドラは `normal` トークンと `power` トークンを取ることで通常状態の船キャラクタとパワーフードの処理であることを表している．
- **R5** は，無敵状態のキャラクタと通常状態のキャラクタ間のルールである．ハンドラは，`unbeatable` トークンと `normal` トークンを取ることでこれらの状態のキャラクタについての処理を表している．
- **R6** は，無敵状態のキャラクタとミサイルの間のルールである．ハンドラは，`unbeatable` トークンと `missile` トークンを取ることでこれらの状態のキャラクタについての処理を表している．

これらのコードを記述から，Join Token 機能を使うことで多数のキャラクタ (オブジェクト) 間の通信に関わる動作が簡潔に表現できることがわかる．具体的には，**図 4.4** は本節冒頭に挙げた複数のキャラクタに関係するルールと直接一対一に対応しており，記述も読解も容易である．

このプログラムには，キャラクタの種類や状態についての条件分岐処理がなく，キャラクタの組み合わせ処理をするループもない．これは，キャラクタの状態による分岐や組み合わせ処理を，ハンドラのパターンマッチによって処理させていることの利点である．さらに，キャラクタ固有の処理と，キャラクタ間の通信を分けて記述できていることも，簡潔さの要因となっている．

4.5.3 シューティングゲーム Balloon

図 4.5 は Join Token と Mogemoge を使用して制作した，前節とは異なるシューティングゲームの画面である [NK11b]．

このプログラムは，画面左下隅にあるプレーヤーの砲台を操作して，画面上部から降りてくる風船爆弾を破壊し，画面下部の都市を守るゲームである．

通常，砲台から発射されるミサイルは風船か爆弾を 1 つしか破壊することができない．しかし，プレーヤーは飛んでいる最中のミサイルを任意のタイ

ミングで爆発させ，爆風を発生させることができる．爆風で破壊された爆弾はさらに爆風を発生させる．爆風により他の爆弾を連鎖して破壊することがこのゲームの面白さになっている．

このゲームのルールを前節のシューティングゲームと同様にまとめたものを以下に示す．

R1. 砲台の向きはプレーヤーが操作する．

R2. プレーヤーはミサイルを発射/爆発させることができる．

R3. 風船は画面上部よりゆっくりと下降してくる．爆弾は各風船の紐の先に結び付けられている．

R4. ミサイルは風船や爆弾を「爆風無しで」破壊することができる．

R5. 紐で結ばれている風船と爆弾のどちらかを破壊すると，もう一方はそのまま落下する．風船だけが破壊された場合，爆弾は通常よりも速く落下する．

R6. ミサイルや爆弾の爆風は複数の風船を破壊できる．

R7. ミサイルや爆弾の爆風は複数の爆弾を破壊できる．

R8. 爆弾と爆風は画面下部の都市の建物を破壊できる．

R9. 爆風を破壊することはできないが，１つの爆風は２つ以上の建物を破壊することはない．

これらのルールも，キャラクタ間の関係を表すルールとそうでないルールの二つに分けられる．キャラクタ間の関係を表すルールは，**R3**, **R4**, **R5**, **R6**, **R7**, **R8**, **R9**. であり，個別のキャラクタのみに関するルールは，**R1**, **R2** である. 前述のシューティングゲーム同様，個別のキャラクタのみに関するルールは，オブジェクトのメソッドで平易に記述可能である．一方，キャラクタ間の関係を表すルールを実装するのに，オブジェクトとメソッドのみで実装するのは容易ではない．しかし Join Token を使用すると，このゲームのキャラクタ間の関係を表すルールを直接的に記述することが可能である．

次は，風船と爆弾のプログラムの初期化部分である．

```
Balloon = object {
  init = method( x ) {     # 風船初期化処理
      bomb = new Bomb; bomb.init( x, 0 ); # 爆弾生成
      throw balloon(bomb); # 爆弾を引数に風船トークン投入
  }
};
Bomb = object {
  init = method( x, y ) {  # 爆弾初期化処理
    throw bomb;            # 爆弾トークン投入
  }
};
```

風船オブジェクトは初期化時に，結ばれている爆弾も同時に生成する．風船も爆弾も，それぞれが初期化時に，balloon と bomb というトークンをトークンプールに登録する．

他のキャラクタも同様に自分自身を表すトークンを throw する．これらのキャラクタを表すトークンを利用して，オブジェクト間の関係を表すルールを実装している．

衝突や爆発に関係したルールは (**R4**, **R5**, **R6**, **R7**, **R8**, **R9**) である．これらは**図 4.6** のハンドラコード群で実装できる．

ルール **R6** と **R7** に対応するハンドラは* を使用して explosion トークンが残るように実装している．これにより，爆風は複数の風船や爆弾を破壊することができる．ルール **R9** に対応する最後に記述されているハンドラは，explosion トークンがトークンプールから削除される例外ルールである．この記述により，爆風はビルを 1 つ破壊するとそれ以降他のキャラクタを破壊できなくなる．

風船と爆弾を結ぶルールである **R3** は次のハンドラにより実装されている．

```
# rule R3: 爆弾は風船に結び付けられている
join *bln.balloon(child) *b.bomb() where child == b {
    b.set_pos( bln.tipx, bln.tipy );
};
```

このハンドラは，balloon と bomb トークンがトークンプールにあり，さらに balloon トークンの引数が bomb と関連しているオブジェクトと一致する場合に起動する．balloon トークンの引数は，風船の初期化をしたときに同時に生成した爆弾オブジェクトであるので，このハンドラは，結ばれている風船と爆弾のトークンの組み合わせについてのみ実行される．

このハンドラの中では，風船の紐の先の位置を表す tipx と tipy の位置に爆弾の位置を設定しており，結果として，結ばれている爆弾は常に風船の紐の先に位置するようになる．

このハンドラはトークンを削除しないので，風船や爆弾が破壊されて対応するトークンがトークンプールから削除されない限り (アニメーションフレームごとに) 実行され続ける．

ルール **R5** は， ルール **R4**，**R6**, **R7** のハンドラで実装されている．例えば，風船がミサイルに衝突した場合，風船の速度が次のコードで再設定され，落下速度が増す．

```
bomb.set_vel( 0, 2 ); # 爆弾の落下速度を変更
```

一方，爆弾とミサイルが衝突した場合には，特に速度の再設定がされないため，残された風船の落下速度が変化することはない．

このゲームのプログラムからも，複数のキャラクタに関係するルールが Join Token によるキャラクタ間の通信として簡潔に実装できることがわかる．ハンドラ内での処理もシンプルで読みやすく，Join Token がゲームルールの実装に適していることがわかる．

```
# rule R4+R5: ミサイルは風船を破壊できる
join m.missile() bln.balloon(bomb) where m.is_collided( bln ) {
  m.destroy(); bln.destroy();  # ミサイルと風船を破壊
  bomb.set_vel( 0, 2 );        # 爆弾の落下を速くする
}
# rule R4(+R5): ミサイルは爆弾を破壊できる
join m.missile() b.bomb() where m.is_collided( b ) {
  m.destroy(); b.destroy();    # ミサイルと爆弾を破壊
}
# rule R6(+R5): 爆風は風船を破壊できる
join *e.explosion() bln.balloon(bomb) where e.is_collided( bln ) {
  bln.destroy();               # 風船を破壊
  bomb.set_vel( 0, 2 );        # 爆弾の落下を速くする
}
# rule R7(+R5): 爆風は爆弾を破壊できる
join *e.explosion() b.bomb() where e.is_collided( b ) {
  b.destroy();
  e = new Explosion;           # 爆風の発生
  e.init( b.x, b.y, 1.2 );     # 爆風の位置設定
}
# rule R8: 爆弾は建物を破壊できる
join b.bomb() bld.building() where b.is_collided(bld) {
  bld.destroy(); b.destroy();  # 建物と爆弾を破壊
}
# rule R8+R9: 爆風は建物を 1 度だけ破壊できる
join e.explosion() bld.building() where e.is_collided(bld) {
  bld.destroy();               # 建物を破壊
}
```

図 4.6: Balloon ゲームの衝突に関係するルールに対応したハンドラ記述

4.5.4　アクションゲーム Descender

図 4.7 も Join Token と Mogemoge を使用して制作した，アクションゲームである [NK11b].

このゲームは前述した二つのシューティングゲームよりもルールが複雑である．プレーヤーの目的は，縦横二種類のロープを使い分けながら，二つのビルの間を降りていくことである．

プレーヤーは動きながら，水平ロープを反対側のビルに向けて張ったり，垂直ロープを延長することができる．

ゲーム中には，降下を邪魔する二種類のキャラクタが登場する．１つは鳥で，時々現れては，プレーヤーに向かって爆弾を落としてくる．プレーヤーが爆弾に当たると，ゲームオーバーとなる．

もう一つはビルの住人である．住人はプレーヤーに直接的なアクションをしない代わりに，張られている垂直ロープを切ろうとする．プレーヤーが掴んでいる垂直ロープが切られるとゲームオーバーである．プレーヤーが掴んでいなくとも，プレーヤーは切られたロープを延長しなければ切られた部分より下に降りることができない．

このゲームのルールを次にまとめる．ルール **R6** 以外は全て Join Token ハ

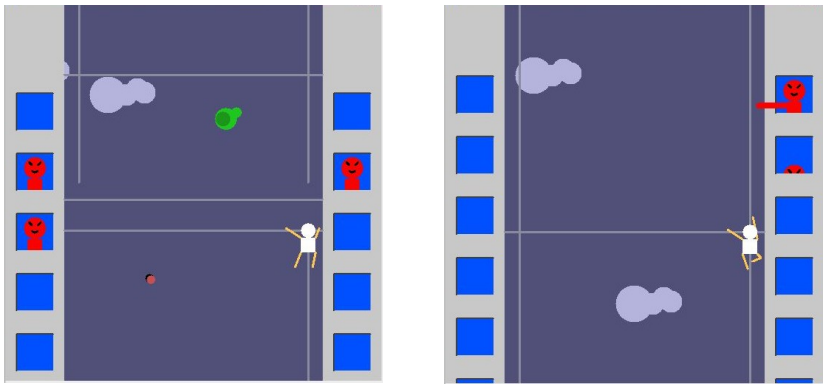

図 4.7: Descender ゲームの画面

ンドラで記述されている．

R1. プレーヤーは現在の場所から，水平ロープか垂直ロープに沿って移動できる．

R2. プレーヤーが垂直ロープの下端にいる場合，プレーヤーはそのロープを下方へ延長することができる．

R3. プレーヤーが下降するとそれに従って他のキャラクタは画面上部へと移動する．プレーヤーの Y 座標は移動しないが，結果としてプレーヤーが下へ降りているように見える．

R4. 雲は他のキャラクタよりもゆっくりと画面上部へ移動する．雲はビジュアル上の効果のためのキャラクタで，他のキャラクタに何か影響を及ぼすことはない．

R5. プレーヤーが垂直ロープに捕まっていて，近くに水平ロープが無ければ，プレーヤーは水平ロープを反対側に張ることができる．

R6. 時々鳥が飛んできて爆弾を落とす．

R7. いくつかのビルの窓にはビルの住人がいて，時々目の前の垂直ロープを切ろうとする．

R8. 爆弾がプレーヤーに当たると，プレーヤーは落ちてゲームオーバーとなる．

R9. プレーヤーが掴んでいる垂直ロープをビルの住人が切ると，プレーヤーは落ちてゲームオーバーとなる．

R10. ビルの住人に切られなければ，垂直ロープは無限に下に伸びていることにする．

ゲーム中，プレーヤーは次の 3 つのモードのどれかの処理をする．

- 垂直移動中 (降下中)
- 水平移動中
- 落下中 (ゲームオーバー時)

プレーヤーのアクションはこのモードに依存しているため，個々のモードに対応するメソッドとして実装されている．各モードの処理はプレーヤーキャラクタの update メソッド内から適宜呼び出されるようになっており，簡単には次のようになっている．

```
Player = object {
  update_descending = method() { ... }
  update_moving_horizontally = method() { ... }
  update_falling_horizontally = method() { ... }
  update = update_descending;   # 最初のモード
}
```

update_で始まるメソッドが個々のモードに対応したメソッドである．例えば，垂直移動中のメソッドは次の実装となっている．

```
update_descending = method() {
  if ( guiKeyPressed( KEY_DOWN ) ) {
    throw cmd_descend;
  }
  if ( guiKeyOn( KEY_LEFT ) ) {
    throw cmd_shoot_hrope;
  } elif ( guiKeyOn( KEY_RIGHT ) ) {
    throw cmd_shoot_hrope;
  }
  is_left_side = ( x == LEFT_PLAYER_X );
  if (guiKeyPressed( KEY_RIGHT ) and is_left_side) {
    throw cmd_go_side;
  } elif (guiKeyPressed( KEY_LEFT ) and not is_left_side) {
    throw cmd_go_side;
  }
};
```

垂直移動中のモードにおいては，プレーヤーは垂直ロープに沿って降下するか，水平ロープを張るかのどちらかのアクションを行える．このようなプレーヤーのアクションは，Descender ゲームプログラムでは，cmd_で始まる名前のトークンとして実装されている．例えば，降下するコマンドは cmd_descend トークン，水平ロープを張るコマンドは cmd_shoot_hrope トークン，水平移動コマンドは cmd_go_side トークンをトークンプールに throw することで，ハンドラがそれぞれの処理を行う．

例えば，降下もしくは垂直ロープを延長することについてのルールである **R1** と **R2** の処理は，次の 2 つのハンドラが行う．

```
join p.cmd_descend *r.v_rope where r.is_on( p.x, p.y ) {
  d = min( r.by - p.y, 2 );
  if ( d > 0 ) {       # 降下 (R1)
    throw scroll( d );
    p.anim_descend();
  } else {             # 垂直ロープを延長する (R2)
    r.extend_to_bottom();
  }
};
join p.cmd_descend { };  # cmd_descend を削除するだけ
```

v_rope は，垂直ロープが自分の存在を示すためにトークンプールに throw するトークンである．

一つ目のハンドラは，プレーヤーが垂直移動コマンドトークンを throw していて，且つ，垂直ロープが存在し，その垂直ロープの上にプレーヤーがいる場合に起動される (上に乗っているかどうかは where 節で検査されている)．垂直ロープのトークンには*がついており，トークンプールから削除されない．

一方，垂直移動トークン cmd_descend には*がついていないので，このハンドラが処理されると削除される．プレーヤーが降り続けるには，プレーヤーが毎フレーム cmd_descend を throw し続けなければならない．

このハンドラ内部では，次の二つの処理をしている．

- プレーヤーの下にロープが伸びているなら，降下処理.
- プレーヤーの下にロープが伸びていないなら，垂直ロープを延長する処理.

前者はルール **R1** に対応し，他のオブジェクトを画面上部へ動かすスクロール処理と，降下するアニメーションの処理をする．スクロールは scroll トークンを throw することで，後述する別に記述されたハンドラにより処理される．トークンを処理の指示に使用するこの実装は cmd_に類似している．

後者は **R2** に対応し，プレーヤーは降下するコマンドで，降下とロープの延長の両方が可能となっている．

二つ目のハンドラは特に内部では何も処理せず，cmd_descend トークンを無条件に削除している (*指定が無い)．これにより，処理されなかった cmd_descend トークンはプール内に残らないので，垂直移動コマンドが次のフレーム以降に持ち越されることがない．

他のプレーヤーのアクションコマンドも，垂直移動コマンドに類似した方法でハンドラにより実装されている．あるフレームで throw されたプレーヤーのコマンドは，そのフレーム内でのみ処理され，それ以降のフレームに影響を及ぼすことはない．

ゲーム画面は，プレーヤーが降下するに従って上方へスクロールする (ルール **R3**)．実際にはこの処理は，プレーヤー以外のキャラクタを全て上方へ移動させることで実装されている．スクロール処理は，scroll トークンが存在するときに実行される．scroll トークンは，前述の垂直移動処理ハンドラ

```
# bg_object を投げるオブジェクト全てを画面上方向へ移動
join *any.scroll(d) *o.bg_object {
  o.y = o.y - d;
  if ( o.y < SCROLL_OUT_LIMIT_Y ) { o.destroy(); }
};
# 垂直ロープのスクロールについての特殊ルール
join *any.scroll(d) *r.v_rope {
  if ( r.y > 0 or d > 0 ) { #
    r.y = r.y - d;
    if ( r.y < 0 ) { r.y = 0; }
  }
  if ( r.by < HEIGHT ) {
    r.by = r.by - d;
    if ( r.by < 0 ) { r.destroy(); }
  }
};
# 雲のスクロール処理
join *any.scroll(d) *c.cloud {
  c.y = c.y - d / 2.0;
};
# 次のフレームに影響しないようにトークンを削除
join any.scroll { };
```

図 4.8: スクロール処理をするハンドラの記述

中に throw されるトークンで，プレーヤーのアクションを表すコマンドと同様に実装されている．スクロール処理を行うハンドラ群は**図 4.8** のとおりである．

スクロールするキャラクタは，bg_object トークンを初期化時にトークンプールに throw する．従って，**図 4.8** の最初のハンドラは，scroll トークンが存在するときに，全ての bg_object に対して実行される．このハンドラは bg_object を投げた全てのキャラクタの Y 座標値を scroll トークンの引数分だけ変化させる．

二番目のハンドラは，垂直ロープに関する特殊なルール **R10** に関連するものである．垂直ロープは，スクロールに関して次の二つのケースがある

- 垂直ロープの端点が画面上端もしくは下端まであるときは，そのまま．
- 垂直ロープの端点が画面端にないときは，その端点はスクロール移動する．

三番目のハンドラは，ルール **R4** の実装で，雲の移動に関するものである．雲は他のキャラクタの半分の速度で上方向へ移動する．

最後のハンドラは，未処理の scroll トークンを削除するハンドラである．トークンの削除のみを目的としているので，内部では何もしていない．プレーヤーのコマンド処理と同様，あるフレームに throw されたトークンは，そのフレームでのみ有効である．

ルール **R5** は**図 4.9** のハンドラ群で実装されている．

```
# 近くにアニメーション中の水平ロープがあるなら,
# 新しく水平ロープを張ることはできない.
join p.cmd_shoot_hrope *r.h_rope_flying
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
# 近くに水平ロープが張られているなら,
# 新しく水平ロープを張ることはできない.
join p.cmd_shoot_hrope *r.h_rope
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
# 水平ロープを張る
join p.cmd_shoot_hrope {
  hr = new HorizontalRope;
  if ( p.x == LEFT_PLAYER_X ) {
    hr.init( p.y, false );
    p.anim_shoot_hrope( false );
  } else {
    hr.init( p.y, true );
    p.anim_shoot_hrope( true );
  }
};
# 次のフレームに影響しないようにトークンを削除
join p.cmd_shoot_hrope { };
```

図 4.9: 水平ロープを張る処理のハンドラの記述

水平ロープは h_rope トークン (反対側に張られていてプレーヤーが掴める) か，h_rope_flying トークン (反対側へ張るアニメーションの最中であるので掴めない) のどちらかを throw する

水平ロープはルール **R5** により，近くに水平ロープが無いときにのみ張ることができる．これは，**図 4.9** の最初の二つのハンドラにより記述されている．この二つのハンドラは，水平ロープを張るコマンド cmd_shoot_hrope トークンがあっても，プレーヤーとの Y 座標の差が BREADTH_TO_CHANGE_ROPE 未満の水平ロープ (もしくは水平に張っている最中のロープ) がある場合，なにもせずに cmd_shoot_hrope トークンを削除している．

図 4.9 の三つ目のハンドラが，実際に水平ロープを張る処理をするハンドラである．このハンドラは，**R5** を満たすケースでは，先の二つのハンドラにより cmd_shoot_hrope が削除されてしまうため，起動されない．

四つ目のハンドラは他のコマンド処理をするハンドラと同様，そのフレームの cmd_shoot_hrope トークンを削除する．

ルール **R7** にあるように，ビルの住人は垂直ロープを切ろうとする．ビルの住人は「本体」と「腕」の二つのキャラクタから成り，垂直ロープを切るのは「腕」の方である．

腕キャラクタは，垂直ロープを切れるだけのところまで伸びると，cmd_cut_rope トークンを throw する．このトークンは，腕キャラクタの切るというアクションを表すトークンで，他のプレーヤーコマンドと同様に throw されたフレー

ムのみ有効なコマンドである．

垂直ロープを切る処理を記述したハンドラは次の二つである (can_cut メソッドは，腕が指定のロープを切れる位置にいるかどうか検査するメソッドである)．

```
# 垂直ロープを切る
join a.cmd_cut_rope *r.v_rope where a.can_cut( r ) {
    r.cut( a.y );
};
# 次のフレームに影響しないようにトークンを削除
join a.cmd_cut_rope { };
```

垂直ロープのメソッド cut はロープを切る処理を行う．このメソッドは垂直ロープを切ったことを cut_vrope トークンを throw することで通知する．この通知は，ルール **R9** を記述するハンドラで利用される．このルールは，切られて落ちるロープにプレーヤーが掴まっている場合，プレーヤーも落下させる処理で，ハンドラの実装は次のようになっている．

```
join *p.player rope.cut_vrope( by, cut_y )
    where p.x == rope.x
      and rope.y <= p.y and p.y <= by
      and cut_y < p.y {
    p.update = p.update_falling;
};
join any.cut_vrope { };
```

ハンドラは，cut_vrope トークンのロープが切られた通知を受け，且つ where 節でプレーヤーがその切られたロープ上にいるかどうかを検査して，両方とも満たされるなら，プレーヤーを落下させる処理を行う．

cut_vrope トークンも他のコマンド同様，throw されたフレームより後に影響が出ないようにするために，二つ目のハンドラで削除されている．

ルール **R8** は前節のシューティングゲームと同じく，衝突時に発生する処理である．そのため対応するハンドラもプレーヤーと鳥の爆弾のトークンがあって，且つ where 節で衝突を検査し，両方とも満たされるなら，プレーヤーを落下させる処理を実行する．このハンドラは次のようになっている．

```
join *p.player o.bitch where p.is_collided( o ) {
    p.update = p.update_falling;
};
```

このように，Descender ゲームは前節二つのゲームよりも複雑なルールのゲームとなっているが，Join Token を使うことにより，列挙したルールがほぼそのまま簡潔に記述できている．

4.5.5　Join Token を使用しないプログラムとの比較

本節では，Join Token をより詳しく評価するために，Join Token 機能の無いプログラミング言語を使用して前節のゲームプログラムを製作し，両者

の比較を行う [NK11b]．比較に際しては，2.6.1 節で議論した，キャラクタ間の通信を表現するための通信モデルとして望まれる次の 4 つを評価の基準とした．

- 理解しやすく通信が簡潔に記述できること．
- 複数の並行処理間の通信の記述に適していること．
- キャラクタの種類や状態に依存した処理が記述に適していること．
- 処理順序やタイミングを明示できること．

プログラミング言語には Ruby[まつ 99] を使用した．Ruby はオブジェクト指向の汎用プログラミング言語である．Ruby は次の理由から比較対象として選択した．

- クラスをベースとしたオブジェクト指向言語で，ビデオゲーム開発で主に用いられる C++言語と基本的な構造が同等であること．
- キャラクタ間の通信以外は Mogemoge に類似したプログラムになると予想されること．
- Tk 等のライブラリが提供されていて，グラフィクスや入力処理の実装がシンプルになること．

Ruby はクラスをベースとしてプログラムを記述する言語であることから，ビデオゲーム開発で主に用いられている C++言語と基本的な構造が同等で，Ruby との比較は，現状のゲーム開発における記述との比較とみなすことができる．

また，Ruby は Join Token を除いては Mogemoge にも近く，キャラクタ間の通信以外について，Ruby と Mogemoge のプログラムは類似したものとなり，比較がしやすいと予想される．

さらに，Ruby には，グラフィクスやデバイス入力を処理するのに便利な Tk[Ous95] 等のライブラリがある，これらを利用することで制作が容易になり，Mogemoge との比較が難しいほどプログラムが複雑になることも無い．

比較対象とするゲームには，キャラクタ間の関係を表すルールが Descender ゲームのように複雑ではなく比較がしやすいことと，Shooting ゲームのように通信が衝突だけではないことから，Balloon ゲームを選択した．

制作した Ruby/Tk の Balloon ゲームプログラムは，Mogemoge によるプログラムの行数は 357 行であるのに対して，Ruby によるプログラムは 436 行となった．Mogemoge と Ruby の文法的な違いはあるが，キャラクタの定義等は類似したものとなっており，違いの多くは，グラフィクスの処理とキャラクタ間の関係を表すルールの処理部分となった．

Balloon ゲームのルールは，風船と爆弾を結びつけるルール以外がキャラクタ同士の衝突として定義されている．そこで，Ruby プログラムでは，キャラクタの衝突検査をする部分のみを `check_collision` メソッドとして切り出し，次のように複数の異なる衝突検査処理で利用可能なように記述した．

```
def check_collision( c1, c2 )
    o1s = $obj_list.find_all { |o| o.kind_of? c1 }
    o2s = $obj_list.find_all { |o| o.kind_of? c2 }
    o1s.each do |o1|
        o2s.each do |o2|
            yield o1, o2 if o1 != o2 && o1.is_collided( o2 )
        end
    end
end
```

Ruby プログラムではキャラクタは全て Ruby のクラスとして定義されている．この check_collision メソッドは，衝突検査を行いたいキャラクタの種類を表すクラスを引数にとり，その種のキャラクタ同士の衝突検査をし，衝突しているなら，このメソッドに渡されるブロックに処理をさせる．

このメソッドを利用することで，衝突に関するルールを処理する Ruby プログラムは**図 4.10** のように，Balloon ゲームの Mogemoge プログラムに近い記述となった．

各 check_collision メソッドの呼び出しは，4.5.3 節の Mogemoge による Balloon ゲームの衝突を処理するプログラムのハンドラ列と対応している．プログラムは上から順に実行されるので，処理順序も，Mogemoge プログラムと同じになっている．

Ruby のメソッドには，Join Token のハンドラの where 節のような，起動条件を記述する方法がない．そこで，Balloon ゲームのハンドラの where 節は，メソッド内で if 文を用いた条件分岐処理で実装した．

Balloon ゲームの，風船と爆弾を結びつけるルール **R3** だけは，衝突に関する処理ではないことから，上記の check_collision メソッドを利用して記述することができない．そこで，Ruby プログラムでは風船と爆弾を相互参照により直接関連付け，ルール **R3** を実装した．次は，その実装部分で，風船オブジェクトの毎フレームの処理部分である．風船は@bomb という変数で自分が結ばれている爆弾を記録している．爆弾も同様に結ばれている風船を参照している．

```
def update
    :
  if !@bomb.nil?
    @bomb.set_pos(@tipx, @tipy) # 爆弾の位置更新
  end
    :
end
```

この実装の if 文は，結び付けられた爆弾が破壊されていないかどうかを検査している．爆弾の破壊を処理するメソッドには次の処理を記述している．

```
def destroy
  # 親の風船の自分への参照を解消する.
  @parent.bomb = nil if !@parent.nil?
end
```

風船の破壊を処理するメソッドも同様に記述した．このメソッドにより，結ばれている相手が破壊されている場合，参照は何も指さない nil となるので，

```
# rule R4+R5: ミサイルは風船を破壊できる
check_collision( Missile, Balloon ) do |m,bln|
  m.destroy; bln.destroy
  bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
end
# rule R4(+R5): ミサイルは爆弾を破壊できる
check_collision( Missile, Bomb ) do |m,b|
  m.destroy; b.destroy
end
# rule R6(+R5): 爆風は風船を破壊できる
check_collision( Explosion, Balloon ) do |e,bln|
  bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
  bln.destroy
end
# rule R7(+R5): 爆風は爆弾を破壊できる
check_collision( Explosion, Bomb ) do |e,b|
  b.destroy()
  Explosion.new( b.x, b.y, 1.2 )
end
# rule R8: 爆弾は建物を破壊できる
check_collision( Building, Bomb ) do |bld,b|
  bld.destroy; b.destroy
end
# rule R8+R9: 爆風は建物を 1 度だけ破壊できる
check_collision( Explosion, Building ) do |e,bld|
  if e.active then e.active = false; bld.destroy end
end
```

図 4.10: Ruby による Balloon の衝突に関するルールの記述

相手の存在を確認でき，爆弾だけが破壊され風船が残った場合でも，破壊さらた爆弾について操作をすることが無いようにできる．

Mogemoge プログラムと Ruby プログラムの Balloon ゲームプログラムは次の点で異なっている．

- Ruby プログラムには衝突を処理を記述しやすくするために，`check_collision` というメソッドを導入している．Join Token では，発火時のハンドラとトークンのマッチングとし処理されるので，このようなメソッドは必要ない．
- Ruby プログラム中では衝突検査をするキャラクタの種類 (クラス名) を指定する．これはゲーム中のキャラクタの状態を表すものではなく，トークン名によって状態を表現可能な Join Token を使用したプログラムとは異なる．
- 風船とミサイルが衝突したときに爆弾の速度を再設定する処理を記述するために，Ruby プログラムでは，風船オブジェクトに関連付けられている爆弾オブジェクトを，風船オブジェクトから直接取得している．Mogemoge プログラムではトークンの引数として表現できるので，このような風船に爆弾を参照させる直接的な関連付けが必要ない．

- Mogemoge プログラムではハンドラの `where` 節で細かい条件を指定するが，Ruby プログラムは衝突処理内部で `if` 文を使用して条件判定を記述する．

  `where` 節はハンドラが通信を処理する条件を明示する．`where` 節を満たさない場合，ハンドラは起動されず，トークンの削除も行われないので，ゲーム進行にも影響が無い．

  一方，Ruby プログラムの `if` 文は衝突処理の起動条件を明示していない．`if` 文を含むメソッド内のコードを全て確認しなければ，条件が成立しないとき，何も処理されないことを知ることができない．

- Ruby プログラムでは，あるキャラクタが複数のキャラクタに衝突して複数のルールが適用可能な場合の記述が難しい．爆風は 1 つの建物しか破壊できないルール **R9** は，Ruby プログラムでは爆風に `active` という変数を導入することで実装している．Mogemoge プログラムでは，**R9** は，単純に爆風のトークンを取り除く (*を指定しない) だけで記述できている．

- 風船と爆弾を結びつけるルール **R3** は衝突ではないため，上記の `check_collision` メソッドを利用して記述することができず，相互参照により解決している．

  この実装には，結びつきを実装するのに相手が破壊されたかどうかの検査が必要で，破壊時には参照を解消する記述も必要となっている Join Token を用いた場合，この参照はトークンの引数として表現可能で，相互参照よりも簡潔に記述でき，バグの発生も回避できる．

これらの違いを元に，2.6.1 節で議論したキャラクタ間の通信を表現するための通信モデルとして望まれる 4 つの性質について，次のことが評価できる．

- Join Token は，簡潔でわかりやすくキャラクタ間の通信を記述できる．
- Join Token は，複数の並行処理間の通信の記述が容易である．
- Join Token は，キャラクタの種類や状態に依存した通信の記述に適している．
- Join Token は，通信の処理順序やタイミングを明示できる．

Ruby プログラムでは，衝突の処理を簡潔にするためは，`check_collision` というメソッドを用意している．Balloon ゲームよりも複雑なルールのゲームシステムを記述する場合，`check_collision` に類似したメソッドが多く必要となると考えられる．また，風船と爆弾の結びつきを実装するには，相互参照を用いた実装が必要となり，記述は簡潔ではない．Join Token を用いた Mogemoge プログラムの場合，`check_collision` に相当するコードは不要であり，風船と爆弾の結びつきも，ハンドラ 1 つで記述できる．このことから，

Mogemoge プログラムは Ruby プログラムよりも，キャラクタ間の通信を簡潔でわかりやすく記述可能であることがわかる．

Ruby プログラムでは，`check_collision` メソッドは，複数のキャラクタ間の通信 (衝突) に関する処理に必要な処理である．また，ruby プログラムでは，爆風は 1 つの建物しか破壊できないというルールの記述のために `active` という変数が必要となっている．Join Token を使用した記述では，どちらも不要であり，トークンの組み合わせ処理や削除で容易に記述できる．このことから，Join Token は複数のキャラクタ間の通信の記述に適していると評価できる．

Ruby プログラムの `check_collision` メソッドを用いた衝突処理は，衝突の記述にキャラクタの種類 (クラス名) しか使えないが，Join Token では，トークン名やトークンの引数で状態を表現可能である．よって，Join Token は，Ruby よりもキャラクタの種類や状態に依存した通信の記述に適している．

Ruby プログラムでは，衝突処理の順序については Join Token と同様に記述可能だが，風船と爆弾の結びつきについてのルールの処理は，順序やタイミングを明示できない．Join Token では，このルールについてのハンドラも，他の衝突を処理するハンドラと同じく記述順に処理させる記述ができるので，通信の内容や実装方法に依存せず，処理順やタイミングを明示することが可能である．

上記以外に，Join Token には実行時の効率性の改善が記述性に影響を与えないという特徴がある．Ruby プログラムではキャラクタの組み合わせを処理するために，階層的なループを使い記述しているが，このループが実行時の効率性などで問題になった場合，ゲームシステム処理部分の可読性は効率性のための修正によりさらに低下する．Mogemoge と Join Token を利用する場合，効率性の問題は Join Token の処理内部を改善すればよく，Join Token を使って記述されたゲームシステムの記述に修正の必要がない．

4.6 まとめ

本章では，新しいゲームキャラクタ間の通信モデルである Join Token を提案した．

Join Token は 3 章で説明した S540 の通信モデルである Tuple Space の問題点を解決するために考案した新しい通信モデルである．

Tuple Space の Tuple は識別する方法がなく，上書きが難しいことから，Join Token は，名前と関連付けられたキャラクタで識別を可能にすることで，書き込み時に古いものを自動的に削除し上書き機能を実装可能なトークンを通信に使用する．

トークンは，Tuple Space 同様，トークンプールというグローバルな場所に保持され，複数のトークンに対して起動されるハンドラを用いてキャラクタ間の通信を表現する．ハンドラは，記述順が処理順序であることから，処理順序を記述できないという Tuple Space の問題を解決し，通信に必要なトーク

ンをトークンプール内から識別するトークンパターンにより，Tuple Space では記述が煩雑になる多数のキャラクタの通信も平易に記述できるようにした．

この，Join Token の機能を評価するために，Join Token を実装したプログラミング言語 Mogemoge を開発し，Mogemoge を使い 3 つのサンプルゲームを制作した．サンプルゲームの制作から，Join Token は，異なるゲームの様々なキャラクタ間の通信を，Join Token を用いて簡潔に記述でき，処理順序も適切に指定可能であることを示し，Tuple Space の問題点が Join Token では解決できていることを示した．

また，Join Token 機能を持たないプログラミング言語 Ruby で 3 つのゲームのうちの 1 つを実装し，Join Token を使用しないプログラムとの比較も行った．この結果，Ruby プログラムよりも Join Token を用いた Mogemoge プログラムの方が，キャラクタ間の通信を明示的にわかりやすく記述でき，2.6.1 節で議論した，ゲームキャラクタ間の通信を表現するための通信モデルとして望ましい性質を持つと評価することができた．

しかし，本章だけでは Join Token が十分に評価できているとは言えない．例えば，Mogemoge 上に Join Token を実装した理由は，Join Token のみの評価を行いたいことと，Join Token の構文や実装を簡潔にするためであったが，他のプログラミング言語や S540 での Join Token の実装については行っていない．Mogemoge は状態遷移処理，時間に沿った処理，並行処理に適した記述機能を持たないので，1 章で述べたゲームシステム記述ツールの要件を満足したツールにまでは至っていない．

また，評価のために制作したゲームは規模も小さく，数も多くない．Join Token の更なる評価と改良のためには，他の種類のゲームへの適用とさらなる評価が必要である．

第5章　結論

ビデオゲーム開発では,「面白さ」の定義が難しいことから実装前に十分な設計をすることが難しいため，トライアンドエラーを繰り返しながら開発が進められる．しかし，近年の大規模化による高度で複雑なプログラムや制作作業量の増大は，トライアンドエラーを難しくしており,「面白さ」を十分に達成できなくなってきている．そこで現在のビデオゲーム開発では，ゲームシステム記述ツールを導入してゲームデザイナーだけで面白さの中核となるゲームシステムの記述を可能し，トライアンドエラーを容易にする開発手法がとられている．

しかし，現在のゲームシステム記述ツールは，ゲームデザイナーがゲームシステムを独自に実装する上で十分な機能を提供していない．本論文は，十分な機能を持ったプログラミング言語の設計と実装を行い，実際にゲーム開発に適用することを通じて評価し，ビデオゲーム開発プロセスの改善に貢献することを目的とした．

1 章では，現在のビデオゲーム開発の現状について詳細に説明した後，ゲームの面白さの中核となる「ゲームシステム」について定義した．このゲームシステムは，グラフィカルインターフェースや数値設定では十分な記述が難しいことから，ゲームシステム記述ツールはプログラミング言語を提供する必要があることを述べた．さらに，そのプログラミング言語は，ゲームデザイナーがプログラマーの協力無しにゲームシステムの記述ができるよう,「状態遷移処理」,「時間の流れに沿った処理」,「並行処理」,「キャラクタ間の通信の 4 つを容易に記述できる必要があると仮定し，これを要件として検証を進めたことについて説明した．

2 章では，現在までに用いられている既存のゲームシステム記述ツールを,「ビデオゲーム開発を目的としたプログラミング言語」,「統合的なビデオゲーム開発環境」,「個々のゲームの開発専用に作られた記述ツール」,「類似するその他のツール」の 4 つに分類し，各記述ツールの特徴や機能について説明した．さらに，各記述ツールについて，1 章で議論した 4 つの要件に従って評価を行った．この評価から，既存の記述ツールは，ゲームシステムの記述に便利な機能を持つものもあるが，1 章で導き出した機能に関する 4 つの要件を十分に満足していないことを明らかにした．

また調査から，既存の記述ツールの多くは要件の 1 つである，ゲームキャラクタ間の通信を記述する機能に欠けていることが判明した．そこで，ゲームシステムにおけるキャラクタ間の通信にどのようなものがあるかについて議論し，計算機科学分野でゲームキャラクタ間の通信を表現するのに利用可

能と考えられる通信モデルについての調査を行った．キャラクタ間の通信についての議論と調査から，各通信モデルについての評価を行い，ゲームシステム記述への適用可能性について議論した．

3 章では，1 章と 2 章の議論や調査を元に設計及び開発した新しいプログラミング言語 S540 とその記述ツールについて説明した．S540 は,「状態遷移処理」,「時間の流れに沿った処理」,「並行処理」,「キャラクタ間の通信」の 4 つの要件を満たすよう設計したプログラミング言語である．S540 は，4 つの要件の他に，開発時の言葉を記述にそのまま用いることができるよう日本語によるプログラムの記述機能を持ち，記述ツールは，ゲームシステム全体を記述可能とするためのツール構造で実行時の処理時間のコストを最小限とする仕組みとなるよう設計した．

さらに 3 章では，この設計に従い開発した S540 記述ツールを，実際のビデオゲーム開発の現場でプロトタイプゲームの開発に使用し評価を行ったことについて説明した．この結果から，S540 を導入することで，ゲームデザイナーがプログラマーの助け無しにゲームシステムの実装を行えたことや，それまでのビデオゲーム開発と比較してトライアンドエラーが容易になったことがわかり，本研究の目的がある程度達成されていることを確認した．また，このことから，1 章で議論した 4 要件を満たすことで，トライアンドエラーが容易になるということも間接的ではあるが確認することができた．

一方で，キャラクタ間の通信のために S540 に導入した通信モデル Tuple Space は，シンプルなモデルであることから理解が容易で，様々な通信が記述可能ではあるが，通信が冗長な記述や煩雑な操作となる場合があり，間違いの発生原因となりやすいことが明らかになった．また，通信の処理順序やタイミングを明示的に記述できず，ゲームバランス上問題になる場合があることもわかった．

この結論から，4 章では，Tuple Space では冗長もしくは煩雑な記述になる場合でも簡潔な記述が可能で，通信の処理順序やタイミングを明示可能な新しい通信モデル Join Token を提案した．Join Token は，キャラクタの状態を表現するトークンとトークンを保持するトークンプール，トークンプール上のトークンの組み合わせから処理を行うハンドラで構成される．Join Token では，キャラクタがどのような状態であるかはトークンから判別でき，通信の処理はハンドラとしてキャラクタの処理とは分離して記述できることから，個々の通信処理が明示的で簡潔に記述でき，さらに通信の処理順序もハンドラの記述順序で明示することが可能である．

この Join Token を評価するために，プログラミング言語 Mogemoge を設計実装し，3 つの異なるゲーム制作に適用した．この結果，異なるゲームの様々なキャラクタ間通信を Join Token により簡潔に記述できることを確認した．また，通信の処理順序も適切に明示することができ，Tuple Space の問題点は Join Token で解決できたことを明らかにした．さらに，Join Token 機能を持たない Ruby でも同一のゲームを制作し，Mogemoge のプログラムと比較することで Join Token が汎用のプログラミング言語と比較してもキャ

ラクタ間の通信を明示的にわかりやすく記述できることを示した．

以上のように本研究では，ビデオゲーム開発におけるトライアンドエラーを容易にするために，ゲームデザイナーが独自にゲームシステムを実装することができる新しいプログラミング言語 S540 とその記述ツールの設計と開発を行い，ゲーム開発に適用して評価することで，ビデオゲーム開発プロセスの改善に貢献できる結果を得ることができた．評価からは，S540 のキャラクタ間の通信を記述するために採用した Tuple Space に記述性の問題があることが明らかになったが，この問題を克服するために新しい通信モデル Join Token を提案し，その実装と評価を行うことで，問題点が解決できることを示した．

また，3 章でゲームデザイナーのみでゲームシステムが記述可能であったことと，4 章で記述性の改善が確認できたことから，4 つの要件を満たすことが，本論文の目的を達成することになることも示すことが出来た．

一方，3 章の S540 については，導入による具体的な効果を検証するために必要な，導入前後の比較検証ができていない．これは，実験に協力いただいたビデオゲーム開発会社が，アクションゲームの開発を行うために新しく設立された企業であり，記述ツール導入前の開発経歴がないことが理由である．開発者らにはビデオゲーム開発の経験があることから，主観的な比較は可能であったが，評価として十分ではない．

また，S540 のキャラクタ間通信の記述機能に問題があったことから，Join Token を提案及び実装し，その評価を行ったが，適した構文で Join Token を記述できるようにするため構文から修正可能なプログラミング言語 Mogemoge に実装したため，1 章で述べた 4 つの機能的要件を全て満たす記述ツールの開発までには至っていない．

さらに，ビデオゲームシステムをゲームデザイナーでも記述可能とするためにプログラミング言語に求められる機能が，前提とした「状態遷移処理」,「時間の流れに沿った処理」,「並行処理」,「キャラクタ間の通信」の 4 つ要素の記述性だけで十分かどうかは，本論文では明らかすることができていない．

このように，本研究にはまだ課題や問題点が残されている．今後は残された課題や問題点の解決に取り組むため，さらに研究を進めていきたいと考えている．

謝辞

本論文は，多くの方々のご指導，ご助言，ご協力によりまとめることができました．ここにご紹介できない方々含め，全ての方々に謹んで感謝の意を表します．

筑波大学大学院ビジネス科学研究科，大木敦雄准教授，吉田健一教授には，本論文と研究について親切なご指導とご助言をいただきました．厚く感謝いたします．また，本論文について多くの有益なコメントをくださった同ビジネス科学研究科の論文審査委員の先生方に感謝いたします．

同大学院久野ゼミの有志皆様からは少なからぬ助言をいただきました．ここに感謝いたします．

1 章では，工学的な観点からだけでなく，経営学的な観点からもゲーム開発の説明をしています．これは筑波大学大学院ビジネス科学研究科経営システム科学専攻で得られた知見や，同期生らの助言を受けたものです．ここに感謝いたします．

2 章の 2.4 節「個々のゲーム開発専用に作られた記述ツール」は，ビデオゲーム業界において最前線でご活躍されている方々にご協力をいただいてまとめることができました．その中から，お名前の記載をご快諾いただいた方々のみ，順不同で次に記させていただきます．ご所属が併記されている方は，本論文執筆時点のものを記載しています．

株式会社ドリームファクトリー 代表取締役 石井精一氏，株式会社スクウェア・エニックス モバイル事業部 シニア・マネージャー 時田貴司氏，株式会社インタラクティブブレインズ 取締役執行役員 笠井秀行氏，株式会社クアッドアロー 代表 小野口正浩氏，株式会社モノリスソフト 開発部 矢島利明氏，株式会社バンダイナムコゲームス 彡湖久治氏，株式会社バンダイナムコゲームス 開発スタジオ コンテンツデザインディビジョン コンテンツデザイン部 第 3 課 石黒正隆氏，株式会社スクウェア・エニックス 第一制作部 開発グループ プランナー 柴田伯一氏，竹内久彦氏．

以上の他にも，2.4 節に関しては，多くのビデオゲーム開発関係者の方々にご協力いただきました．諸事情からご提供，ご協力いただいた全てを本論文中にまとめることはできませんでしたが，ご紹介した皆様と関係者の方々には深く感謝いたします．

3 章の S540 と S540 記述ツールを使ったプロトタイプゲームの開発は，株式会社娯匠 (http://www.goshow.co.jp/) において実施されました．当時，会社はまだ立ち上げたばかりで運営も手探り状態にあった中，この S540 という，全く新しいプログラミング言語を用いたゲーム開発に興味を示され，使

用をご快諾いただいた堀内雅史社長に感謝いたします.

プロトタイプゲームは，ゲームデザイナーである，渋谷哲也氏，猪俣猛氏，前野信太郎氏の 3 氏に S540 記述ツールを使用していただきながら開発が進められました．3 氏からは，未経験の開発方法で試行錯誤する中，多大なるご助言とご協力をいただきました．氏らの協力なくして，本論文が完成することはありませんでした．S540 は，Scripting language の頭文字「S」と,「娯匠」の読みである「ごしょう」を数字で表記した「540」が名前の由来ですが,後ろの 540 という表記部分は猪俣氏によるものです．また，3.5.6 節のインタビュー中の「BASIC のようだった」は渋谷氏,「体で覚えた」は前野氏のコメントであります．さらに，同じくゲームデザイナーである，石川将光氏，井澤一海氏からは，多くの有益なご助言をいただき，4 章に繋がる知見となりました．プロトタイプゲーム開発後の「グラディエーターロードトゥフリーダム」という製品の開発で S540 は引き続き使用され，ゲームは完成して発売に至りますが，この過程においても両氏の協力は欠かせませんでした．S540 については，梅田信太郎氏，伊藤太陽氏，松本隆芳氏，元澤利明氏からもご協力をいただきました.

以上の，娯匠のメンバーと，娯匠での S540 を用いた開発に携わられた方々に厚く感謝いたします．方々との S540 を使ったゲーム開発プロジェクトは,現在までの著者の経験の中でも最も楽しかったものの 1 つとなっています.

4 章の 3 つのゲーム，Shooting，Balloon，Descender は主に，NOVA2001(ユニバーサルプレイランド社 1984 年)，バルーンボンバー (タイトー社 1980 年),ミサイルコマンド (アタリ社 1980 年)，CRAZY DESCENDER(電波新聞社月刊マイコン 1982 年 9 月号掲載) に影響を受けたゲームとなっています．これらは，設置店が遠い，子供時分に遊びにいける場所に無い，該当機種を十分触る機会が無い，といった理由で，著者には遊びたくてもなかなか遊べなかったゲームです．過去には決して社会的に十分認められた娯楽とは言えなかったビデオゲームについて，それを主題とした本論文をまとめることができたのは，これらのゲームを含めた先人方の創造の積み重ねがあったからだと思います.

本論文では述べていませんが，情報処理振興事業協会 (IPA) の平成 15 年度未踏ソフトウェア創造事業の援助を受けて，S540 を更に修正したプログラミング言語の開発を行いました．プロジェクトマネージャーのラムダ数学研究所代表 伊知地宏氏には，少なからぬ示唆をいただけたことに感謝いたします．(S540 の) 良いところは泥臭いことを技術的な美しさにとらわれず素直に泥臭くやっているところ，というお言葉は，現在のプログラミングについての著者の取り組み方ともなっています.

1 章については，妻にも意見をもらいました．また，2.5 節の,「プログラミン」については，プログラミンで遊ぶ子供たちからも知見を得ることができました．81 ページ**図 2.25** は長男 (当時 3 歳) が作っていた画面です．夜は家にいなかったり，休日には論文を書いたりと，迷惑をかけながらも本論文として結実できたのは，家族の協力があったからです．彼らにも心より感謝い

たします．

最後に，大学院修士課程入学時から博士課程まで，長い間ご指導いただいた，筑波大学大学院ビジネス科学研究科，久野 靖教授に感謝いたします．本論文についてはもちろんのこと，久野教授のご指導なくして，現在の技術者，研究者としての著者はありませんでした．進捗は遅々として芳しくなく，出来が良いとは言えない学生であったにも関わらず，長きにわたって本論文と研究だけにとどまらない様々な面での熱心なご教示とご鞭撻をいただきました．ここに重ねて深謝いたします．本当にありがとうございました．

参考文献

[Ado] Adobe. *Adobe Flash Platform*. http://www.adobe.com/.

[Ale93] Repenning Alexander. Agentsheets: a tool for building visual programming environments. *CHI 92 Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems*, pp. 142–143, 1993.

[BK09] Gary Bradski and Adrian Kaehler. 詳解 OpenCV – コンピュータビジョンライブラリを使った画像処理・認識. オライリージャパン, 2009.

[Boe00] James Boer. オブジェクト指向プログラミングと設計テクニック, 第 1.1 章. Game Programming Gems 日本語版. ボーンデジタル, August 2000.

[BPT10] Christos J. Bouras, Vassilis Poulopoulos, and Vassilis Tsogkas. Squeak etoys interactive and collaborative learning environments. *Gaming And Simulations: Concepts, Methodologies, Tools and Applications*, pp. 898–909, November 2010.

[BS98] Frederic Boussinot and Jean-Ferdy Susini. The Sugarcubes tool box: a reactive java framework. *Software: Practice and Experience*, Vol. 28, No. 14, pp. 1531–1550, 1998.

[CAB+00] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, pp. 486–493. ACM, 2000.

[CD95] CONITEC-DATASYSTEMS. 3d Game Studio A5, 1995. `http://www.conitec.net/index.htm`.

[ces03] CESA ゲーム白書 2003. CESA, July 2003.

[ces09] CESA ゲーム白書 2009. CESA, July 2009.

[CHFL11] Michael Carter, Martin Hunt, Tom Fairfield, and Jacob Lyles. Gameclosure, 2011. `http://www.gameclosure.com/`.

[Cia94] Paolo Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, Vol. 12, pp. 251–283, 1994.

[Cor] Microsoft Corporation. Microsoft XNA. `http://msdn.microsoft.com/ja-jp/xna/`.

[Cra82] Chris Crawford. *The Art of Computer Game Design*. McGraw-Hill, 1982.

[CRY04] CRYTEK. Cryengine, 2004. `http://www.crytek.de`.

[Daw02a] Bruce Dawson. Game scripting in python, 2002. GDC 2002.

[Daw02b] Bruce Dawson. ゲームオブジェクト AI 用のマイクロスレッド, 第 3.2 章. Game Programming Gems 2 日本語版. ボーンデジタル, June 2002.

[Dem04] Alberto Demichelis. Squirrel, 2004. `http://squirrel-lang.org/`.

[DF05] Danny Dubé and Marc Feeley. Bit: A very compact scheme system for microcontrollers. *HigherOrder and Symbolic Computation*, Vol. 18, No. 3-4, pp. 271–298, 2005.

[DRW95] Andrew Douglas, Antony Rowstron, and Alan Wood. Isetlinda: Parallel programming with bags. Technical report, Department of Computer Science, The University of York, 1995.

[Dyb00] Eric Dybsand. 有限状態マシンクラス, 第 3.1 章. Game Programming Gems 日本語版. ボーンデジタル, August 2000.

[ECM99] ECMA. *ECMAScript language specification 3rd edition*. 1999.

[ECM06] ECMA. *Standard ECMA-334 C# Language Specification 4th edition*. 2006.

[EG02] Epic Games. Unreal Developer Network, 2002. `http://udn.epicgames.com/Main/WebHome.html`.

[Emm09] Paul Emmerich. *Beginning Lua with World of Warcraft Addons*. Apress, July 2009.

[Far04] Charles Farris. 関数ポインタを使った組み込み型有限状態マシン, 第 3.3 章. Game Programming Gems 3 日本語版. ボーンデジタル, December 2004.

[ffc08] 小さな王様と約束の国 ファイナルファンタジークロニクル, 2008. `http://www.square-enix.co.jp/littleking/`.

[FFMS08] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A Language for concurrent distributed and mobile programming. Vol. 2638 of *Lecture Notes in Computer Science*, pp. 129–158. Springer, 2008.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pp. 406–421. Springer-Verlag, 1996.

[FH03] E. Friedman-Hill. *Jess in Action: Rule-Based Systems in Java.* Publications Co., Greenwich, CT, 2003.

[Fje04] Frode Vatvedt Fjeld. Movitz: Using Common Lisp for kernel-level programming on commodity hardware., 2004. Workshop on Evolution and Reuse of Language Specifications for Domain Specific Languages at ECOOP2004.

[Fla07] David Flanagan. *JavaScript.* オライリー・ジャパン, August 2007.

[For82] C. L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artifical Intelligence*, Vol. 19, No. 1, pp. 17–37, 1982.

[FPB02] Jr. Frederick Phillips Brooks. 人月の神話: 狼人間を撃つ銀の弾はない. ピアソンエデュケーション, 2002.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, Vol. 7, pp. 80–112, January 1985.

[Gel89] David Gelernter. Multiple tuple spaces in linda. In *PARLE '89 Proceedings of the Parallel Architectures and Languages Europe*, No. 1 in Lecture Notes in Computer Science, pp. 20–27. Springer, 1989.

[GH98] Etienne M. Gagnon and Laurie J. Hendren. Sablecc – an object-oriented compiler framework. *Proceedings of the Technology of Object-Oriented Languages and Systems*, pp. 140–154, August 1998.

[GHV95] Erich Gamma, Richard Helm, and Ralph Johnson John Vlissides. オブジェクト指向における再利用のためのデザインパターン. ソフトバンク, 1995.

[Gos00] James Gosling. Java 言語仕様第 2 版. ピアソンエデュケーション, December 2000.

[Gre09] Jason Gregory. State-based scripting in uncharted 2: Among Thieves. In *Game Developers Conference(GDC 2009)*, December 2009.

[Gro10] Books Group. *Lua-scripted video games: Freeciv, Grim Fandango, Far Cry, PlayStation Home, World of Warcraft, Ragnarok Online, Empire: Total War, Crysis*. Books LLC, November 2010.

[Har99] Eric Harlow. GTK+・GDK による Linux アプリケーション開発. 翔泳社, 1999.

[Har05] Matthew Harmon. ゲームエンティティ管理システム, 第 1.8 章. Game Programming Gems 4 日本語版. ボーンデジタル, July 2005.

[IdFC05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, Vol. 11, No. 7, pp. 1159–1176, 2005.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua – an extensible extension language. *Software: Practice & Experience*, Vol. 26, pp. 635–652, June 1996.

[igd09] ゲームにおけるスクリプト言語の現状. ゲームテクノロジー研究会 (SIG-GT) 第 12 回研究会. 国際ゲーム開発者協会日本 (IGDA 日本), January 2009.

[IR] projet Moscova INRIA Rocquencourt. The join calculus language. http://join.inria.fr/.

[iS96] id Software. Quake, 1996.

[Jac06] Scott Jacobs. 状態機械のビジュアルな設計, 第 1.15 章. Game Programming Gems 5 日本語版. ボーンデジタル, September 2006.

[Jon01] Andreas Jonsson. Angelscript, 2001. `http://www.nscripter.com/`.

[KR89] Brian W. Kernighan and Dennis M. Ritchie. プログラミング言語 C 第 2 版. 共立出版株式会社, June 1989.

[LEG98] LEGO.com. Mindstorms Home, 1998. http://mindstorms.lego.com/.

[Luc87] LucasArts. Scumm, 1987. `http://hak.wablog.com/75.html`.

[Lut98] Mark Lutz. Python プログラミング. オライリージャパン, 1998.

[MBK+04] J Maloney, L Burd, Y Kafai, N Rusk, B Silverman, and M Resnick. Scratch: a sneak preview. *Proceedings Second International Conference on Creating Connecting and Collaborating through Computing 2004*, pp. 104–109, 2004.

[MCa] Microsoft Corporation. Microsoft Excel.

[MCb] Microsoft Corporation. Microsoft Visual Studio. `http://www.microsoft.com/japan/msdn/vstudio/`.

[Mon96] Olivier Montanuy. *QuakeC Specifications v1.0 (unofficial)*, 1996. `http://www.gamers.org/dEngine/quake/spec/quake-spec34/qc-menu.htm`.

[Mor09] Shun Moriya. スクリプト言語 mana, 2009. `http://mana.sourceforge.jp/ja-jp/`.

[MP05] Mica and Philippe. Flashdevelop.org, 2005. `http://www.flashdevelop.org/`.

[MT01] Motion-Twin. Mtasc, 2001. `http://www.mtasc.org/`.

[NB95] Marc A. Najork and Marc H. Brown. Obliq-3d: A high-level, fast-turnaround 3d animation system. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, pp. 175–193, June 1995.

[NK11a] Taketoshi Nishimori and Yasushi Kuno. Join token: A language mechanism for programming interactive games. *Entertainment Computing*, 2011. (採録予定).

[NK11b] Taketoshi Nishimori and Yasushi Kuno. Join Token-Based Event Handling: A Comprehensive Framework for Game Programming. In *Software Languages Engineering*, Vol. 6940 of *Lecture Notes in Computer Science*. Springer, 2011. (採録予定).

[Ous95] John K. Ousterhout. Tcl&Tk ツールキット. ソフトバンククリエイティブ, 1995.

[Pal05] Mike Pall. LuaJIT, 2005. `http://luajit.org/`.

[Par01] Miles T. Parker. What is ascape and why should you care? *J. Artificial Societies and Social Simulation*, Vol. 4, No. 1, 2001.

[Rab04] Steve Rabin. AI エージェント、AI オブジェクト、AI クエスト用の拡張可能なトリガーシステム, 第 3.5 章. Game Programming Gems 3 日本語版. ボーンデジタル, December 2004.

[RD03] Matthew Riek and Greg Douglas. GameMonkeyScript, 2003. http://www.somedude.net/gamemonkey/.

[Rie99] Thiadmer Riemersma. The small scripting language. In *Dr.Dobb's Journal*. CMP Media LLC, October 1999.

[Roc04] Rockstar Games. Grand theft auto: Sanandreas, 2004. http://www.rockstargames.com/sanandreas/.

[RW98] Antony Rowstron and Alan Wood. Solving the linda multiple rd problem using the copy-collect primitive. *Sciences of Computer Programming*, Vol. 31, pp. 335–358, July 1998.

[SCE09] Sony Computer Entertainment. Uncharted 2: Among Thieves, 2009.

[SMK09] Václav Slováček, Miroslav Macík, and Martin Klíma. Development framework for pervasive computing applications. *SIGACCESS Access. Comput.*, pp. 17–29, September 2009.

[SOF] ALICE SOFT. System 3.x/System 4. http://www.alicesoft.com/support/sys35.html.

[SR08] Rich Shupe and Zevan Rosser. 初めての ActionScript3.0. オライリー・ジャパン, August 2008.

[SS98] Dimitrios Souflis and Jonathan S. Shapiro. Tinyscheme, 1998. http://tinyscheme.sourceforge.net/.

[Str98] Bjarne Stroustrup. プログラミング言語 C++第 3 版. Addison-Wesley Publishers Japan Ltd., November 1998.

[Sus06] Jean-Ferdy Susini. The reactive programming approach on top of java/j2me. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pp. 227–236. ACM, 2006.

[syl07] Sylphis3d, 2007. http://devnet.sylphis3d.com/.

[SZ02] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, September 2002.

[Tec06] Unity Technologies. Unity 3, 2006. http://unity3d.com/.

[Tis00] C. Tismer. Continuations and Stackless Python or "how to change a paradigm of an existing program". *Proceedings of the 8th International Python Conference*, 2000.

[UBI04] UBISOFT. Far cry, 2004. http://farcry.uk.ubi.com/.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pp. 227–242. ACM, 1987.

[VR02] Alex Varanese and John Romero. *Game Scripting Mastery*. Premier Press Inc., December 2002.

[W3C11] W3C. *HTML5 Working Draft 25*. May 2011.

[WCO02] L. Wall, T. Christiansen, and Jon Orwant. プログラミングPerl. オライリー・ジャパン, August 2002.

[W.De03] W.Dee, PIA 少尉. 吉里吉里／ＫＡＧではじめるゲーム制作. 工学社, May 2003.

[win01] Win32 API オフィシャルリファレンス改訂3版 グラフィック/GUI 編. アスキー書籍編集部, 2001.

[まつ 99] まつもとゆきひろ, 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. アスキー出版局, November 1999.

[アー 05] アーテイン. グラディエーターロードトゥフリーダム, 2005. http://www.ertain.com/ja/products/gg/gladiator/.

[エン 04] エンターブレイン. RPG ツクール, 2004. http://tkool.jp/.

[ソニ 96] ソニー・コンピュータエンタテインメント. クラッシュバンディクー, 1996. http://www.jp.playstation.com/software/title/scps10031.html.

[兼宗 01] 兼宗進, 御手洗理英, 中谷多哉子, 福井眞吾, 久野靖. 学校教育用オブジェクト指向言語「ドリトル」の設計と実装. 情報処理学会論文誌 プログラミング (PRO), Vol. 42, No. SIG11 PRO12, pp. 78–90, 2001.

[高橋 09] 高橋直樹. Nscripter, 2009. http://www.nscripter.com/.

[斎藤 09] 斎藤実, 須藤崇夫, 堀口真史. StarLogo プログラミング. 東京電機大学出版局, October 2009.

[小野 01] 小野田洋仁郎. Black diamond, 2001. http://www.onoda-pro.com/black/project/bd/index.htm.

[松原 03] 松原義継. あるビデオゲームの開発事例—納期優先に基づく行動及び開発経費の比較. 赤門マネジメント・レビュー, Vol. 2(10), pp. 531–538, 2003.

[松田 09] 松田白朗. ゲーム機向けの ECMA Script 実装「CriScript」. ゲームテクノロジー研究会 (SIG-GT) 第 12 回研究会. 国際ゲーム開発者協会日本 (IGDA 日本), January 2009. http://www.cri-mw.co.jp/event/2009/sig-gt-12.html.

[新清 02] 新清士. ゲーム開発最前線「侍」はこうして作られた－アクワイヤ制作 2 課の 660 日戦争. 新紀元社, 2002.

[新宅 03] 新宅純二郎, 田中辰雄, 柳川範之. ゲーム産業の経済分析. 東洋経済新報社, February 2003.

[星一 08] 星一. Ruby のゲーム開発の現状と自作ゲームライブラリ Star Ruby. 2008. RubyKaigi 2008.

[清木 04] 清木昌. 自動検証機能を備えたゲームシナリオ記述用開発環境. 第 45 回プログラミング・シンポジウム報告書, Vol. 45, pp. 97–106, 2004.

[西森 03] 西森丈俊, 久野靖. アクションゲーム記述に特化した言語. 情報処理学会論文誌 プログラミング (PRO), Vol. 44, No. SIG15 PRO19, pp. 36–54, 2003.

[石橋 07] 石橋立宣. xtal-language, 2007. http://code.google.com/p/xtal-language/.

[薦田 97] 薦田憲久, 安信千津子, 大川剛直. エキスパートシステムの設計と開発. 昭晃堂, 1997.

[竹田 00] 竹田陽子. プロダクト・リアライゼーション戦略. 白桃書房, 2000.

[中谷 02] 中谷多哉子, 兼宗進, 御手洗理英, 福井眞吾, 久野靖. オブジェクトストーム：オブジェクト指向言語による初中等プログラミング教育の提案 (< 特集 > オブジェクト指向技術). 情報処理学会論文誌, Vol. 43, No. 6, pp. 1610–1624, 2002.

[長慎 01] 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. 情報処理学会夏のプログラミング・シンポジウム報告集, pp. 99–103, 2001.

[湯浅 03] 湯浅太一. Java アプリケーション組込み用の lisp ドライバ. 情報処理学会論文誌 プログラミング, Vol. 44, No. 4, pp. 1–16, 2003.

[白石 08] 白石史明, 土田俊郎. Wii ware project lifecycle: Final fantasy crystal chronicles, the little king. Game Developers Conference 2008, February 2008.

[文部 10] 文部科学省. プログラミン, 2010. http://www.mext.go.jp/programin/.

[湊 10] 湊和久, 宇藤慶彦. 案ずるより産むがやすし!xtal と gamemonkeyscript によるガンシュー開発. CEDEC2010, 2010. http://cedec.cesa.or.jp/2010/program/PG/C10_P0320.html.

[悠黒 01] 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.

付 録A　プログラミング言語S540の資料

A.1　プログラミング言語S540の構文の仕様

拡張バッカスナウア記法 (拡張 BNF) による，プログラミング言語 S540 の構文の仕様を掲載する．バッカスナウア記法 (BNF) は，プログラミング言語 Algol60 の構文を記述するために用いられた．拡張 BNF はこれを拡張した記法である．

拡張 BNF による構文の記述は，生成規則と呼ばれる文の構成の仕方を記述した規則の集まりからなる．例えば，

式 ::= 数字列 | 数字列 '+' 数字列

という生成規則は，式が、「数字列」のみか，2 つの数字列を「+」記号でつないだもので構成されることをあらわす．生成規則は再帰的な記述も可能である．

式 ::= 数字列 | 式 '+' 数字列

この生成規則により,「数字列 + 数字列 + 数字列 + ...」という記述も式として定義できる．この構文仕様中で使用している記号を，**表 A.1** に示す．

表 A.1: 記号の意味

記号	意味
::=	定義．右辺が生成規則で左辺が規則名
|	生成規則の選択
$[X]$	空，もしくは X
$\{X\}$	空または 1 回以上の X の繰り返し
$(X\|Y)$	X または Y
'XYZ'	XYZ という文字の列そのもの
改行	プログラムテキストの各行の行末，もしくはテキストの終わり

前後を引用符 (') でくくられた記号のうち，アルファベットで表記されているものは，大文字と小文字を区別しない．例えば，'iF'，'If'，'if' は全て'IF' と同じ記号として扱われる．また，同じ文字であれば，ASCII 文字と日本語文字 (例えば'A' と' A' 等) も区別しない．

ただし，この文字についての規則は，生成規則中の「文字列リテラル」の引用符"の内部には適用されない．

この構文仕様において，プログラム全体をあらわす開始記号は「プログラム」である．

```
プログラム ::= { プログラム要素 }
プログラム要素 ::= キャラクタ定義 | ライブラリ関数定義
キャラクタ定義 ::= 識別子 改行 変数宣言並び スレッド並び 終わり 改行
変数宣言並び ::= { 識別子 ':' 型 改行 }
型 ::= 'INT'|'REAL'|'STRING'|'CHARACTER'|'VECTOR'|'BOOL'|
       '整数'|'実数'|'文字列'|'キャラクタ'|'ベクタ'|'論理'
スレッド並び ::= スレッド { スレッド区切り 改行 スレッド }
スレッド ::= 状態 { 状態 }
状態 ::= '@' 識別子 改行 文並び
文並び ::= { 文 改行 }
文 ::= if 先頭 { elif 節 } [ else 節 ] 終わり |
  ('WHILE'|'条件ループ') 式 改行 文並び 終わり |
  ('TIMES'|'ループ') 式 改行 文並び 終わり |
  ('INFLOOP'|'無限ループ') 改行 文並び 終わり |
  ('SYNC'|'同期') | ('BREAK'|'ループ脱出') |
  ('WAIT'|'待つ') 式 | 'GOTO' '@' 識別子 |
  'CHANGE' '@' 識別子 |
  ('ERASE'|'消す') 式 |
  ('OUT'|'書く') '(' Tuple データ並び ')' |
  ('IN'|'取る') '(' Tuple データ並び ')' |
  ('RD'|'読む') '(' Tuple データ並び ')' |
  ('EACH'|'組み合わせ') 識別子 カンマ {識別子} 改行 文並び 終わり |
  ('PRINT'|'プリント') 引数並び | 識別子 ':' 型 | 式
if 先頭 ::= 'IF' 式 改行 文並び
elif 節 ::= 'ELIF' 式 改行 文並び
else 節 ::= 'ELSE' 改行 文並び
Tuple データ並び ::= 識別子 [ カンマ Tuple 引数並び ]
Tuple 引数並び ::= Tuple 引数 { カンマ Tuple 引数 }
Tuple 引数 ::= 式 | '?' | '?' 識別子 | '?' 識別子 ':' 型
式 ::= 論理和式 '=' 論理和式 | 論理和式
論理和式 ::= 論理和式 'OR' 論理積式 | 論理積式
論理積式 ::= 論理積式 ('AND'|'且つ') 論理否定式 | 論理否定式
論理否定式 ::= 'NOT' 比較式 | 比較式
比較式 ::= 加減算式 '>' 加減算式 | 加減算式 '>=' 加減算式 |
  加減算式 '<' 加減算式 | 加減算式 '<=' 加減算式 |
  加減算式 '==' 加減算式 | 加減算式 '!=' 加減算式 | 加減算式
加減算式 ::= 加減算式 '+' 項 | 加減算式 '-' 項 | 項
```

```
項 ::= 項 ('*'|'×') 冪乗項 | 項 ('/'|'÷') 冪乗項 | 項 '%' 冪乗
項 | 冪乗項
冪乗項 ::= 冪乗項 ('**'|'××') 単項式 | 単項式
単項式 ::= '-' ベクタ要素式 | ベクタ要素式
ベクタ要素式 ::= 関数呼び出し式 左カギ括弧 式 右カギ括弧 |
                 関数呼び出し式
関数呼び出し式 ::= [ 因子 小数点 ] 識別子 '(' 引数並び ')' | 因子
因子 ::= 実数リテラル | 整数リテラル | 識別子 | 文字列リテラル |
  ('SELF' | '自分') |
  ('CREATE'|'作る') 識別子 |
  ('TRUE'|'真'|'正しい') |
  ('FALSE'|'偽'|'間違い') |
  '(' 式 ')' |
  '(' 式 カンマ 式 カンマ 式 ')' |
  ('RDP'|'読んでみる') '(' Tuple データ並び ')' |
  ('INP'|'取ってみる') '(' Tuple データ並び ')'
ライブラリ関数定義::=
  'EXTERN' 型 識別子 文字列リテラル '(' [ 型並び ] ')'
引数並び ::=  | 式 { カンマ 式 }
型並び ::= 型 { カンマ 型 }

スレッド区切り ::= '===' { '=' }
整数リテラル ::= 数字 { 数字 }
実数リテラル ::= { 数字 } 小数点 { 数字 }
識別子 ::= (英字|仮名|漢字) { (英字|仮名|漢字) }
文字列リテラル ::=  '"' {"以外の文字} '"'

小数点 ::= '.' | '。'
カンマ ::= ',' | '、'
終わり ::= 'END' | '終わり' | 'おわり' | 'オワリ' | '終り'
左カギ括弧 ::= '[' | '「' | '『'
右カギ括弧 ::= ']' | '」' | '』'
```

A.2　導入時の入門用テキスト

ここで挙げるのは，記述ツール導入時に配布した入門用のテキストである．テキスト内には，開発プロジェクトに関する固有の言葉や，俗語等があったため，それらについては修正を行っている．

キャラクタ

プログラムはキャラクタの「定義」集まりです。

```
例）「西森」というキャラクタの「定義」をする
    西森
      @処理すること
        プリント　"怠慢プログラマー"
    終わり
```

当然、キャラクタはいくつでも定義できます。

```
例）「西森」と「にしもり丸」を定義する
    西森
      @処理
        プリント　"普通のプログラマー"
    終わり

    にしもり丸
      @処理
        プリント　"もひかんプログラマー"
    終わり
```

実は、上記のプログラムをコンパイルしても動くプログラムにはなりません。それは次を読みましょう。

「定義」を書いているということは、「キャラクタ」そのものではないことに注意してください。実際に動くキャラクタを登場させるには「キャラクタを作る」という作業が必要になりますが、これはまた後で出てきます。

プログラムはどこから実行されるのか？

プログラムを実行すると「Ｒｏｏｔ」というキャラクタを一つ作って、それが動き出す仕組みになっています。

```
例）実行すると"もげもげ"と表示するプログラム
    Ｒｏｏｔ
      @処理
        プリント　"もげもげ"
    終わり
```

"プリント"はコンソールに文字や数値を表示する命令です。"@処理"とありますが、これは次の「状態」で説明します。

状態

キャラクタは実は様々な状態になり、様々な処理をします。

```
例）食べたり歩いたりする。
  Ｒｏｏｔ
    @食べる
      プリント　"むしゃむしゃ"
      待つ　５
      ジャンプ　@歩く
    @歩く
      プリント　"ちょーちょーちょーいいかんじ♪"
      待つ　１０
      ジャンプ　@食べる
  終わり
```

上記の例では、最初「@食べる」のプログラムが実行され、そのあと「@歩く」が実行されます。「待つ　５」というのは「５フレーム待つ」という意味です。「ジャンプ　@歩く」は状態を「@歩く」にする、という意味です。(@歩くがないとコンパイルエラーになります)。

@の後につける名前は基本的に自由です。ただし、最初に動き出すのは一番先頭に書いたところ（状態）であることは覚えておいてください。また、一つのキャラクタの中で同じ名前の「状態」を書くことはできません。

- Ｒｏｏｔが最初に作られる。
- 作ったＲｏｏｔが最初に動き出す

と覚えてください。ちなみに、Ｒｏｏｔがないプログラムをコンパイルすると、コンパイルエラーになります。　　あと、「終り」「終わり」「おわり」「オワリ」はどれでも大丈夫です。ただし、「尾張」はダメです。

ちなみに、アルファベットの大文字小文字は無視されます。また、全角文字でも半角文字にあるものは半角文字として処理されます。

- ｙ→Y
- Ａ→A
- ａ→A
- ＋→+

並列な処理

キャラクタはいろいろな処理を並列にしたいこともあるでしょう。

例）３フレームおきに３回"ビーム"と表示
　　５フレームおきに２回"爆発"と表示

```
Ｒｏｏｔ
  @処理１
    プリント　"ビーム"
    待つ　３
    プリント　"ビーム"
    待つ　３
    プリント　"ビーム"
    待つ　３
  ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
  @処理２
    プリント　"爆発"
    待つ　５
    プリント　"爆発"
    待つ　５
終り
```

「＝＝＝＝」で切ると、それ以下はまったく別の平行に処理されるプログラムとなります。「＝＝＝＝」は実は長さは４文字以上の「＝」であればいくらでもかまいません。「＝＝＝＝」で平行な処理をいくつでもつくれます。

例）いろいろな挨拶をいろいろな時間間隔で２回づつ表示

```
Ｒｏｏｔ
  @処理１
    待つ　３
    プリント　"おはよう"
    待つ　３
    プリント　"おはよう"
  ＝＝＝＝＝
  @処理２
    待つ　１０
    プリント　"こんにちは"
    待つ　１０
    プリント　"こんにちは"
  ＝＝＝＝＝
  @処理３
    待つ　１５
    プリント　"さようなら"
    待つ　１５
    プリント　"さようなら"
終り
```

たとえ「＝＝＝＝」で切っても状態の名前（@のあとの名前）に同じ名前を使うことはできません。

変数と型

BASIC 等でのいわゆる「変数」を作ってみます。

```
例）全員の年齢の総和を計算してみよう
  Ｒｏｏｔ
    @処理
      山田：整数
      田中：整数
      佐藤：整数
      加藤：整数
      全員：整数

      山田＝２５　　＃２５歳
      田中＝２７　　＃２７歳
      佐藤＝１０３　＃１０３歳
      加藤＝３３　　＃３３歳

      全員＝山田＋田中＋佐藤＋加藤
      プリント　"全員の年齢を足すと"、全員、"歳です"
  終り
```

そんなに難しくはないですね。余談ですが、「プリント」は「、」や「,」で切ると一つ以上のものを表示することができます。

もう少し正確に言えば変数を使いたいときは、使う前に以下のように書かなければならない、ということです。

```
「変数の名前」:「変数の型」
```

変数の型には以下のようなものがあります。サンプルで例を出します。

```
例）いろいろな変数の型
  Ｒｏｏｔ
    @処理
      西森の年齢：整数　　＃これは年齢だから整数でいいよね
      西森の体重：実数　　＃体重は１．５とかもあるだろうし
      西森の何か：キャラクタ　＃これはキャラクタを入れる変数

      西森の年齢＝３３
      西森の体重＝６３．５
      プリント　"１歳あたりの体重＝"、西森の体重÷西森の年齢
  終り
```

「キャラクタ」という変数「西森の何か」がありますが、ここでは、こういう変数も作れるのだな、と覚えておけば結構です。他にも型はありますが、必要に応じて追々説明します。

変数はどんな場所で使えるのか？

定義しない変数は当然つかえません。しかし、定義しても使える場所と使えない場所があります。

```
例）どの変数がつかえないのだろうか？
  Ｒｏｏｔ
    体重：整数        #（A）

    @処理１
      年齢：整数      #（B）
      年齢＝１０      #（C）
      体重＝５５      #（D）

    @処理２
      体重＝３３      #（E）
      年齢＝１５      #（F）!!!!!!?
  終り
```

例について順に説明します。

（A）変数を定義しています。実はキャラクタの定義の先頭でも変数は定義できます。

（B）変数を定義しています。

（C）（B）の変数「年齢」に１０を代入します。これは正しいです。

（D）（A）の変数「体重」に５５を代入にます。これも正しいです。

（E）（A）の変数「体重」に３３を代入します。これも正しいです。

（F）（B）の変数「年齢」に１５を代入します。これは間違いです！

間違いは（F）です。つまり、

- 一つの状態の中に書いた変数はその処理の中でしか使えない。
- キャラクタの定義の先頭に書いた変数はキャラクタ内のどこからでも使うことができる。

前者は状態内でしか使えない変数、後者はキャラクタ内でしか使えない変数、と言う事もできます。

制御構造

ここまでで実はもっとも単純な制御構造文である「待つ」が出ています。

```
-----------------------------------------------
説明：「待つ」
  指定フレーム、時間だけ処理をとめる（１０月７日で単位はフレーム）

  例）待つ　５        ＃５フレーム待ちます
-----------------------------------------------
```

通常のプログラミング言語にある、条件分岐、条件繰り返し、また、特殊なものとして、無限繰り返し、指定回数繰り返し、があります。

```
-----------------------------------------------
説明：「条件分岐」
  条件によって処理を切り替える

  例）年齢３０歳を境に動作が遅くなります。

    Ｒｏｏｔ
      @処理
        年齢：整数
        年齢＝３３     ＃ここを変えると言うことが変わります

        ＩＦ　年齢＜３０
            プリント　"ホップ"
            待つ　３
            プリント　"ステップ"
            待つ　３
            プリント　"ジャンプ!!"
        ＥＬＳＥ
            プリント　"ほっぷ…"
            待つ　１０
            プリント　"ずてっぷ…"
            待つ　１０
            プリント　"じゃんぷぅ…"
        終り

    終り

  例）各年齢層で言うことが違います

    Ｒｏｏｔ
      @処理
        年齢：整数
        年齢＝１５     ＃ここを変えると言うことが変わります

        ＩＦ　年齢＜５
            プリント　"ばぶぅ！"
        ＥＬＩＦ　年齢＜１２
            プリント　"おなかすいたぁ"
        ＥＬＩＦ　年齢＜１８
            プリント　"おい、金！"
        ＥＬＩＦ　年齢＜２２
            プリント　"こんどの合コンさぁ"
        ＥＬＩＦ　年齢＜３０
            プリント　"めっちゃ忙しいって！"
        ＥＬＩＦ　年齢＜５０
            プリント　"娘がさぁ…"
        ＥＬＳＥ
            プリント　"…"
        終り
    終り
-----------------------------------------------
```

```
------------------------------------------
説明：「条件ループ」文
  指定の条件が成立している間、処理を繰り返します。

  例）１０回ループです。

    Ｒｏｏｔ
      @処理
        カウント：整数
        カウント＝０
        条件ループ　カウント＜１０
          プリント　"たつんだ！ジョー！"
          カウント＝カウント＋１
        オワリ
    オワリ
------------------------------------------

------------------------------------------
説明：「無限ループ」文
  無限にループします

  例）５フレームおきに永遠に「テレサー！」と言い続けます

    Ｒｏｏｔ
      @処理
        無限ループ
          プリント　"テレサー！"
          待つ　５
        オワリ
    終わり
------------------------------------------

------------------------------------------
説明：「ループ」文
  指定回数だけ繰り返します。

  例）「条件ループ」文の例とおなじことをします

    Ｒｏｏｔ
      @処理
        ループ　１０
          プリント　"たつんだ！ジョー！"
        終わり
    終わり
------------------------------------------
```

たくさんキャラクタを作る

普通ゲーム中のキャラクタは複数種類あり、同じものでも複数個あることが一般的です。

```
例)「西森」を５フレームおきに３個作る
  Ｒｏｏｔ
    @処理
      作る　西森
      待つ　５
      作る　西森
      待つ　５
      作る　西森
  終わり

  西森
    @処理
      プリント　"うまれたよー"
      待つ　１５
      プリント　"うまれてから１５フレーム経過〜"
  終わり
```

それぞれのキャラクタは「作る」でつくってから独自に動き出します。例では上から順番にキャラクタが定義されていますが、文法上では順番に特に意味はありません。

タプルとタプルスペース

今回のスクリプト言語の目玉商品です。まだ基本的なものしか実装していないため、やや機能として弱い部分がありますが、十分いろいろのことができると思います。

とりあえず、まずは基本的な概念の説明をします。

通信

ゲーム中は各キャラクタは身勝手好き勝手に処理をするのではなくて、なんらかのルールや、待ち合わせ（このフラグが立つまで待つ！とか）で進行します。これらは一般化すると「キャラクタ同士がなんらかの情報交換「通信」をしていて、それにしたがって行動している」と捕らえることができます。

以下にいくつか例を挙げます。

「Ａが死んだらＢが逃げる」 １対１の通信。Ａが死んだときに、Ｂに自分が死んだことを知らせる。

「子分たちが攻撃中はボスは待機」 多対１の通信。子分はそれぞれボスに「自分は攻撃中」と知らせる。ボスは子分の誰か一人でも攻撃中なら待機している。

「相手がすごい攻撃をしようとしたら全員逃げる」 １対多の通信。「すごい攻撃」という情報を多数のキャラクタがいつも見張っている。

「ザコが３人小屋に入ったら、他のザコは入らない」「不特定不定数」対「不特定不定数」の通信。「小屋に入った」という情報をカウントして、３人以上なら小屋に入るのをやめます。

「ある時点でパンチを押したら、あとで２段目に移行」時間的に分離した通信。「ある時点」でパンチを押したら，そのことが後で２段目に移行するときのスイッチになります。

「ある敵を逃したら最後のボスがものすごい攻撃」時間的に分離した通信。「逃げられた」という情報をボスに伝えます。ボスは後からやってくる（か、あとから作られる）ので、自分が行動を始めたときに「逃げられた」という情報が伝わってきたら「すごい攻撃」を始めます。

普通こういったルールは「フラグ」や「グローバルな変数」、「関数（メソッド）呼び出し」や「メッセージ交換」のような様々の手段を使ってプログラムに書くのですが、実はそれは同じ「情報交換」を違う手段で書いている、ということになっています。こういったわけで、誰が考えたのかは知りませんが（実は知ってますが)、「じゃぁ、『通信』という一般的なシステムにしてしまえばいいじゃん」と考えて作られたのが、今回の通信システムらしいです(いや本当はかなり違うみたいですけど)。

板

同じようなシステムで人工知能の分野で「黒板システム」というのがあります。最近のモダンにあわせて（？）とりあえず「板」というと直感的にわかりやすいと思うので以下「板」という言葉を乱発します。まずは例から

```
例)「ばーか」というメッセージを出す処理とそれを受ける処理を並列に動かす
  Ｒｏｏｔ
    @処理１
      待つ　５
      書く（ばーか）　　＃（Ａ）
      待つ　５
      プリント　"おしまい"
    ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
    @処理２
      取る（ばーか）　　＃（Ｂ）
      プリント　"なんだとー！"
  終わり
```

（Ａ）の「書く」は「板に『ばーか』と書く」という処理です。（Ｂ）の「取る」は「板に『ばーか』と書いてあったら消す」という処理です。

処理１は

- ５フレーム待つ
- 「ばーか」と板に書く
- ５フレーム待つ

- 「おしまい」と板に書く

となり、処理２は

- 板に「ばーか」と書かれるのを待って、書かれたら消す。
- "なんだとー！"と表示

となります。もうすこしは面白みのあるサンプルを。

```
Ｒｏｏｔ
  @処理
    作る　どらやき屋
    作る　どらえもん
おわり

どらやき屋
  @処理
    待つ　５
    書く（どらやき、２）
    待つ　５
    書く（どらやき、３）
    待つ　５
    書く（どらやき、１）
おわり

どらえもん
  @処理
    どらやき数：整数
    食べた数：整数

    食べた数＝０
    無限ループ
      取る（どらやき、？どらやき数）　＃（Ｃ）
      食べた数＝食べた数＋どらやき数
      プリント　"合計"、食べた数、"個食べました"
    終わり
おわり
```

この例で板への書き込んでいるものは、最初の引数が「識別」のための名前と、その後の「、」や「,」で区切った通常の式の並びです。

「どらやき屋」は３回どらやきを出します。「どらえもん」はそのだされた「どらやき」を食べていって、今までに食べた合計を表示します。

新しいところは（Ｃ）の部分の「？どら焼き数」でしょう。板に書かれるのは「どらやき、数」です。数は整数を書いているので「どらやき、整数」と考えることができます。一方、「どらえもん」の方では「どらやき、？」という書き込み（？）を待ち続けています。？のところが「どらやき数」で整数ですから、「どらえもん」は「どらやき屋」の「どらやき、整数」という書き込みをちゃんと読むことができる、ということになります。実は、どらえもんの部分だけは以下のように書くこともできます。

```
どらえもん
  @処理
    食べた数：整数

    食べた数＝０
    無限ループ
      取る（どらやき、？どらやき数：整数）　＃（D）
      食べた数＝食べた数＋どらやき数
      プリント　"合計"、食べた数、"個食べました"
    終わり
おわり
```

@処理のすぐ下にあった「どらやき数：整数」がなくなって、かわりに（D）のところに直接書いてあります。意味はどちらでも同じです。　むずかしいですかね？難しかったら西森に聞いてください。もう少しサンプルを出します。

例）さっきの小屋に３人入ったらもう入れないザコ達
　位置とかコリジョンをデータとして今は持てないので、とりあえず、
　「処理が始まって１０フレームたったら小屋に入れる」ということ
　にします。そして、小屋に入れた人は５フレーム後に小屋を出て、
　また１０フレーム待つことにします。入れない人は入れるまで待ちます。

```
Ｒｏｏｔ
  @処理
    ＃板に「小屋にいま０人、入れます」と書きます
    書く（小屋、０、"入れる"）

    ＃５フレームおきにザコを全部で１０人作る
    ループ　１０
      作る　ザコ
      待つ　５
    終わり
終わり

ザコ
  @処理
    人数：整数
    待つ　１０

    ＃小屋に"入れる"まで待つ。入れるなら板から消します
    取る（小屋、？人数、"入れる"）

    ＃小屋の人数を増やして板に書き直します。
    ＃人数が３人になっていたら"入れない"と書きます

    人数＝人数＋１
    ＩＦ　人数＝＝３
      書く（小屋、人数、"入れない"）
    ＥＬＳＥ
      書く（小屋、人数、"入れる"）
    ＥＮＤ

    ＃小屋に入っている間の処理

    プリント　"入りました。今"、人数、"人います"
    待つ　５
    プリント　"出ます。"

    ＃小屋の人数を一人減らして"入れる"にします。

    取る（小屋、？人数、？）　　＃いったん小屋情報を消します。
    書く（小屋、人数－１、"入れる"）　＃出たら絶対"入れる"だよね。

    ジャンプ　@処理　　＃無限ループ
終わり
```

　ここで小屋を二つにしてみます。さすがに入った小屋がどれかわからないとかなるのは問題なので、板への書き込みには「小屋番号」を追加します

例）小屋を２つに増やす
コメントは前の例と同じ物は取ってあります。

```
Ｒｏｏｔ
  @処理
    書く（小屋、０、"入れる"、１）　＃最後のは「小屋番号１」
    書く（小屋、０、"入れる"、２）　＃最後のは「小屋番号２」
    ループ　１０
      作る　ザコ
      待つ　５
    終わり
終わり

ザコ
  @処理
    待つ　１０
    取る（小屋、？人数：整数、"入れる"、？小屋番号：整数）
    人数＝人数＋１
    ＩＦ　人数＝＝３
      書く（小屋、人数、"入れない"、小屋番号）
    ＥＬＳＥ
      書く（小屋、人数、"入れる"、小屋番号）
    ＥＮＤ
    プリント　小屋番号、"に入りました。今"、人数、"人います"
    待つ　５
    プリント　小屋番号、"から出ます。"
    取る（小屋、？人数、？、小屋番号）
    書く（小屋、人数－１、"入れる"、小屋番号）
    ジャンプ　@処理
終わり
```

ここから、「小屋ごとに入れる人数が違う」というのはそれほど難しくないと思います。板への書き込みに「入れる人数」を追加してあげればいいわけです。

実はもっと簡単な全然別の書き方ができます。

例）小屋の人数を板に書く代わりに「小屋チケット」を人数分だけ
板に「貼っておく」ようにする。

```
Ｒｏｏｔ
  @処理
    書く（小屋チケット、１）　＃小屋１の「チケット」を板に貼っておく
    書く（小屋チケット、１）　＃小屋１の「チケット」を板に貼っておく
    書く（小屋チケット、１）　＃小屋１の「チケット」を板に貼っておく
    書く（小屋チケット、１）　＃小屋１の「チケット」を板に貼っておく
    書く（小屋チケット、２）　＃小屋２の「チケット」を板に貼っておく
    書く（小屋チケット、２）　＃小屋２の「チケット」を板に貼っておく
    ループ　１０
      作る　ザコ
      待つ　５
    終わり
終わり

ザコ
  @処理
    待つ　１０

    取る（小屋チケット､？小屋番号：整数）　＃小屋チケットを「取る」
    プリント　小屋番号、"に入りました"

    待つ　５

    書く（小屋チケット、小屋番号）　　＃小屋チケットを「戻す」
    プリント　小屋番号、"から出ました"

    ジャンプ　@処理
終わり
```

板に「小屋チケット」を「張っておく」としています。小屋に入っている人数は情報としてなくなっているので、人数自体を取ることはできませんが（これはまだここまでに説明している機能ではできない)、ほぼ同じ処理ができていることがわかります。

付 録B　プログラミング言語 Mogemogeの資料

B.1　プログラミング言語Mogemogeの構文の仕様

拡張バッカスナウア記法（拡張BNF）による，プログラミング言語Mogemogeの構文の仕様を掲載する．

この構文仕様において，プログラム全体をあらわす開始記号は「プログラム」である．

```
プログラム ::= 文列
文列 ::= { 文 }
文 ::= 'if' '(' 式 ')' '{' 文列 '}' { elif部 } [ else部 ] |
       'while' '(' 式 ')' '{' 文列 '}' |
       'for' '(' 式 ';' 式 ';' 式 ';' ')' '{' 文列 '}'
       'print' 式 ';' |
       'result' 式 ';' |
       'inspect' 式 ';' |
       式 ';' |
       代入文 ';' |
       'throw' 識別子 '(' [ 式列 ] ')' ';' |
       'dispose' 識別子 ';' |
       'ignition' |
       'join 文頭部 '{' 文列 '}

elif部 ::= 'elif' '(' 式 ')' '{' 文列 '}'
else部 ::= 'else' '{' 文列 '}'

代入文 ::= 変数 '=' 式 | 論理和式
変数 ::= 呼び出し式 '.' 識別子 | 識別子

join文頭部 ::= 'join' { トークン } [ 起動条件 ]
トークン ::= ['*'] 識別子 '.' 識別子 '(' [ 識別子列 ] ')'
起動条件 ::= 'where' 式

式 ::= 論理和式
```

```
論理和式 ::= 論理和式 'or' 論理積式 | 論理積式
論理積式 ::= 論理積式 'and' 論理否定式 | 論理否定式
論理否定式 ::= ['!'] 比較式
比較式 ::= bit 演算式 ('<'|'<='|'>'|'>='|'=='|'!=') bit 演算式 |
           bit 演算式
bit 演算式 ::= bit 演算式 ('&'|'|'|'^'>>'|'<<') 加減算式 |
                  加減算式
加減算式 ::= 加減算式 ('+'|'-') 項 | 項
項 ::= 項 ('*'|'/'|'\%') 単項式 | 単項式
単項式 ::= '-' 呼び出し式 | 'new' 呼び出し式 | 呼び出し式
呼び出し式 ::= 呼び出し式 '(' 式列 ')' | 呼び出し式 '.' 識別子 |
                 因子
項 ::= 識別子 | 整数リテラル | 実数リテラル | 文字列リテラル |
       'self' | 'true' | 'false' | 'nil' |
       'exist' 識別子 | 'object' '{' 文列 '}' |
       'method' '(' [ 識別子列 ] ')' '{' 文列 '}' |
       'java' 文字列 '(' [ 文字列並び ] ')' |
       '(' 式 ')'

式列 ::= 式 | 式列 ',' 式
識別子列 ::= 識別子 | 識別子列 ',' 識別子
文字列並び ::= 文字列並び ',' 文字列 | 文字列

整数リテラル ::= 数字 { 数字 }
実数リテラル ::= { 数字 } '.' { 数字 }
文字列リテラル ::= '"' {"以外の文字} '"'
識別子 ::= (英字|'_') { (英字|数字|'_') }
```

B.2 シューティングゲームShootingのMogemogeプログラム

4 章で説明した，3 つのゲームのうち，Shooting の全ソースコードを次に掲載する．

```
WIDTH  = 512;
HEIGHT = 512;
SOFS_X = 3;
SOFS_Y = 5;
PI     = 3.14159265358979;

frame_timer = 0;

#-----------------------------------------------------------------
# キャラクタのベースとなるオブジェクト
Char = object {
  x   = 0; y   = 0;
  vx  = 0; vy  = 0;
  dir = 0; spd = 0;
  init_char = method( _x, _y, _dir, _spd ) {
    x = _x; y = _y;
    set_vel( _dir, _spd );
    ObjectList.add( self );
  };
  update_vel = method() {
    vx = cos( dir ) * spd; vy = sin( dir ) * spd;
  };
  set_vel = method( _dir, _spd ) {
    dir = _dir; spd = _spd;
    update_vel();
  };
  update_position = method() {
    x = x + vx; y = y + vy;
    if ( ( vx < 0 and x < 0 ) or ( vx > 0 and x > WIDTH ) ) {
      vx = -vx;
    }
    if ( ( vy < 0 and y < 0 ) or ( vy > 0 and y > HEIGHT ) ) {
      vy = -vy;
    }
    dir = atan2( vy, vx );
  };
  is_collided = method( obj ) {
    dx = obj.x - x;
    dy = obj.y - y;
    result dx * dx + dy * dy < 256;
  };
};
#-----------------------------------------------------------------
# 船
Ship = Char + object {
  wcr = 128; wcg = 230; wcb = 70;
  id       = 0;
  timer    = 0;
```

```
  ai_timer = 0;
  stat     = 0;
  damage_total   = 0;

  init = method( _x, _y, _dir, _spd, _id ) {
    init_char( _x, _y, _dir, _spd );
    id = _id;
    throw normal;
  };
  update = method() {
    control();
    update_position();
    if ( exist unbeatable ) {
      timer = timer - 1;
      if ( timer < 0 ) {
        dispose unbeatable;
        throw normal;
      }
    }
  };
  set_color = method( r, g, b ) {
    wcr = r; wcg = g; wcb = b;
  };
  cmd_turn_left  = method() { dir = dir - PI / 30; };
  cmd_turn_right = method() { dir = dir + PI / 30; };
  cmd_shot = method() {
    m = new Missile;
    m.init( x, y, dir, spd * 2, id );
    m.r  = wcr * 7 / 5;
    m.g  = wcg * 7 / 5;
    m.b  = wcb * 7 / 5;
  };
  key_control = method() {
    if ( guiKeyPressed( KEY_LEFT ) )  { cmd_turn_left();  }
    if ( guiKeyPressed( KEY_RIGHT ) ) { cmd_turn_right(); }
    if ( guiKeyOn( KEY_SPACE ) )      { cmd_shot();       }
    update_vel();
  };
  ai_control = method() {
    ai_timer = ai_timer - 1;
    if ( ai_timer < 0 ) {
      stat = 0;
      cmd_shot();
    }
    if ( stat == 0 ) {
      ai_timer = rand() * 30 + 30;
      n = rand();
      if ( n < 0.5 )    { stat = 1; }
      elif ( n < 0.75 ) { stat = 2; }
      else              { stat = 3; }
    } elif ( stat == 1 ) {
      ;
    } elif ( stat == 2 ) {
      cmd_turn_left();
    } elif ( stat == 3 ) {
      cmd_turn_right();
```

```
    }
    update_vel();
  };
  control = key_control;
  draw    = method() {
    if ( exist unbeatable and frame_timer % 2 == 0 ) {
      guiDrawShip( x, y , dir, 255, 255, 0 );
    } else {
      guiDrawShip( x, y , dir, wcr, wcg, wcb );
    }
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose normal;
    dispose unbeatable;
  };
  make_unbeatable = method() {
    timer = 200;
    dispose normal;
    throw unbeatable;
  };
  damage = method( d ) {
    damage_total = damage_total + d;
    b = new Bang;
    b.init( x, y );
    if ( damage_total > 50 ) {
      destroy();
    }
  };
};
#-----------------------------------------------------------------
# power food
PowerFood = Char + object {
  init = method( _x,_y ) {
    init_char( _x, _y, 0, 0 );
    throw power;
  };
  update = method() {};
  draw = method() {
    guiDrawPowerFood( x, y, 15, 255, 120, 0 );
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose power;
  };
};
#-----------------------------------------------------------------
# ミサイル
Missile = Char + object {
  timer  = 70;
  id     = 0;
  damage = 5;
  r = 255; g = 128; b = 70;
  init = method( _x, _y, _dir, _spd, _id ) {
    init_char( _x, _y, _dir, _spd );
    id = _id;
```

```
    throw missile;
  };
  update = method() {
    update_position();
    timer = timer - 1;
    if ( timer < 0 ) {
      destroy();
      p = new Pom;
      p.init( x, y );
    }
  };
  draw = method() {
    guiDrawMissile( x, y, dir, r, g, b );
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose missile;
  };
};
#-----------------------------------------------------------------
# 爆発
Bang = Char + object {
  life = 15;
  init = method( _x, _y ) {
    init_char( _x, _y, 0, 0 );
  };
  update = method() {
    life = life - 1;
    if ( life < 0 ) {
      ObjectList.remove( self );
    }
  };
  draw = method() {
    guiSetColor( 180, 0, 0 );
    guiSetLineWidth( 2 );
    guiDrawBang( x, y, 25 );
  };
};
#-----------------------------------------------------------------
# 小爆発
Pom = Char + object {
  timer = 10;
  size  = 1.5;
  init = method( _x, _y ) {
    init_char( _x, _y, 0, 0 );
  };
  update = method() {
    timer = timer - 1;
    if ( timer < 0 ) {
      ObjectList.remove( self );
    }
  };
  draw = method() {
    guiSetColor( 100, 100, 100 );
    guiDrawCircle( x + SOFS_X, y + SOFS_Y, timer * size );
    guiSetColor( 220, 220, 220 );
```

```
    guiDrawCircle( x, y, timer * size );
  };
};
#-------------------------------------------------------------------
# join handlers
# rule R2
join *s1.normal() *s2.normal() where s1.is_collided( s2 ) {
  dx = s2.x - s1.x;
  dy = s2.y - s1.y;
  len = sqrt( dx * dx + dy * dy );
  dx = dx / len;
  dy = dy / len;
  s1.x = s1.x - dx * 8;
  s1.y = s1.y - dy * 8;
  s2.x = s2.x + dx * 8;
  s2.y = s2.y + dy * 8;
  p = new Pom;
  p.init( ( s1.x + s2.x ) / 2, ( s1.y + s2.y ) / 2 );
  p.size = 8.5;
  p.life = 4;
};
# rule R3
join *s.normal() m.missile() where s.id != m.id and s.is_collided( m ) {
  p = new Pom;
  p.init( m.x, m.y );
  s.damage( m.damage );
  m.destroy();
};
# rule R4
join *s.normal() p.power() where s.is_collided( p ) {
  s.make_unbeatable();
  p.destroy();
};
# rule R5
join *s1.unbeatable() *s2.normal() where s1.is_collided( s2 ) {
  s2.damage(1);
};
# rule R6
join *s.unbeatable() m.missile()
       where s.is_collided( m ) and s.id != m.id {
  p = new Pom;
  p.init( m.x, m.y );
  m.destroy();
};
#-------------------------------------------------------------------
# main program

# プレーヤー生成
myship = new Ship;
myship.init( WIDTH / 2, HEIGHT / 2, 0, 4, 0 );

# 敵生成
id = 1;
while ( id < 3 ) {
  e = new Ship;
  e.init( rand() * WIDTH, rand() * HEIGHT, rand() * PI * 2, 3, id );
```

```
  e.control = e.ai_control;
  e.set_color( 230, 200, 200 );
  id = id + 1;
}

# power food 生成
i = 0;
while ( i < 10 ) {
  p = new PowerFood;
  p.init( rand() * WIDTH, rand() * HEIGHT );
  i = i + 1;
}

# メインループ
init_game = method() {
  guiSetClearColor( 180, 180, 180 );
};
update_game = method() {
  ObjectList.iterate( method( e ) { e.update(); } );
  ignition;
};
draw_game = method() {
  guiDrawGrid( WIDTH, HEIGHT, 64 );
  ObjectList.reverse_iterate( method( e ) { e.draw(); } );
};
game_loop( WIDTH, HEIGHT, init_game, update_game, draw_game, 20 );
```

B.3 シューティングゲーム Balloon の Mogemoge プログラム

4 章で説明した，3 つのゲームのうち，Balloon の全ソースコードを次に掲載する．

```
WIDTH  = 512;
HEIGHT = 512;
PI     = 3.14159265358979;
G      = 4.0 / 30;

frame_timer = 0;

#---------------------------------------------------------------------
# キャラクタのベースとなるオブジェクト
Char = object {
  x   = 0; y   = 0;
  vx  = 0; vy  = 0;
  spd = 0;
  diameter = 16;
  init_char = method( _x, _y ) {
    set_pos( _x, _y );
    set_vel( 0, 0 );
    ObjectList.add( self );
  };
  set_pos = method( _x, _y ) {
    x = _x;
    y = _y;
  };
  set_vel = method( _vx, _vy ) {
    vx = _vx;
    vy = _vy;
  };
  update_position = method() {
    x = x + vx;
    y = y + vy;
  };
  is_collided = method( obj ) {
    dx = obj.x - x;
    dy = obj.y - y;
    r = (diameter + obj.diameter) / 2;
    result dx * dx + dy * dy < r * r;
  };
};
#---------------------------------------------------------------------
# 風船
Balloon = Char + object {
  r = 0; g = 0; b = 0;
  swing_cycle = 0;
  swing_angle = 0;
  string_len  = 50;
  tipx        = 0;
  tipy        = 0;
  init = method( x ) {
```

```
    init_char( x, 0 );
    set_vel( 0, rand() * 1.2 + 0.3 );
    r = irand(128) + 100;
    g = irand(128) + 100;
    b = irand(128) + 100;
    radius = 20;
    bomb = new Bomb;
    bomb.init( x, 0 );
    throw balloon(bomb);
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose balloon;
  };
  update = method() {
    swing_cycle = swing_cycle + PI / 40;
    update_position();
    update_tip_pos();
    if ( y > HEIGHT ) {
      destroy();
    }
  };
  update_tip_pos = method() {
    swing_angle = sin( swing_cycle ) * PI / 4;
    tipx = x + sin( swing_angle ) * string_len;
    tipy = y + 8 + cos( swing_angle ) * string_len;
  };
  draw = method() {
    guiSetColor( r, g, b );
    guiDrawCircle( x, y, diameter );

    guiSetColor( 255, 255, 255 );
    guiSetLineWidth( 2 );
    guiDrawLine( x, y + 8, tipx, tipy );
  };
};
#---------------------------------------------------------------------
# 爆弾
Bomb = Char + object {
  init = method( _x, _y ) {
    init_char( _x, _y );
    diameter = 6;
    throw bomb;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bomb;
  };
  update = method() {
    update_position();
    if ( y > HEIGHT ) {
      destroy();
      e = new Explosion;
      e.init( x, y, 1 );
    }
  };
```

```
  draw = method() {
    guiSetColor( 255, 0, 0 );
    guiDrawSquare( x, y, diameter );
  };
};
#-----------------------------------------------------------------
# 風船を作る
BalloonGenerator = object {
  timer = 0;
  update = method() {
    n = 200 - frame_timer / 20;
    if ( n < 50 ) {
      n = 50;
    }
    if ( frame_timer % n == 0 ) {
      b = new Balloon;
      b.init( rand() * WIDTH );
    }
  };
};
#-----------------------------------------------------------------
# ミサイル
Missile = Char + object {
  timer = 0;
  init = method( _x, _y, _vx, _vy ) {
    init_char( _x, _y );
    set_vel( _vx, _vy );
    diameter = 10;
    throw missile;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose missile;
  };
  update = method() {
    vy = vy + G;
    timer = timer + 1;
    if ( timer > 10 and not guiKeyPressed( KEY_SPACE ) ) {
      destroy();
      b = new Explosion;
      b.init( x, y, 0.8 );
    }
    update_position( );
    if ( x > WIDTH or y > HEIGHT ) {
      destroy();
    }
  };
  draw = method() {
    guiSetColor( 0, 255, 255 );
    guiDrawCircle( x, y, diameter );
  };
};
#-----------------------------------------------------------------
# 爆発
Explosion = Char + object {
  timer = 0;
```

```
  init = method( _x, _y, _diameter_scale ) {
    init_char( _x, _y );
    diameter = 100 * _diameter_scale;
    throw explosion;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose explosion;
  };
  update = method() {
    timer = timer + 1;
    if ( timer > 15 ) {
      destroy();
    }
  };
  draw = method() {
    n = ( irand(2) % 2 ) * 100;
    guiSetColor( 180, n, n );
    guiDrawCircle( x, y, diameter );
  };
};
#-----------------------------------------------------------------
# 砲台
Battery = Char + object {
  dir = -PI / 4;
  tx  = 0;
  ty  = 0;
  dx  = 0;
  dy  = 0;
  len = 40;

  init = method() {
    init_char( 0, HEIGHT );
    update_geom();
  };

  update = method() {
    ulim = -PI / 2;
    blim = 0;

    if ( guiKeyPressed( KEY_UP ) ) {
      dir = dir - PI / 40;
      if ( dir < ulim ) {
        dir = ulim;
      }
    } elif ( guiKeyPressed( KEY_DOWN ) ) {
      dir = dir + PI / 40;
      if ( dir > blim ) {
        dir = blim;
      }
    }
    update_geom();

    if ( guiKeyOn( KEY_SPACE ) ) {
      spd = 10;
      missile = new Missile;
```

```
      missile.init( tx, ty, dx * spd, dy * spd );
    }
  };
  update_geom = method() {
    dx = cos( dir );
    dy = sin( dir );
    tx = x + cos( dir ) * len;
    ty = y + sin( dir ) * len;
  };
  draw = method() {
    guiSetLineWidth( 8 );
    guiSetColor( 128, 128, 128 );
    guiDrawLine( x, y, tx, ty );
    guiSetLineWidth( 3 );
    guiSetColor( 200, 200, 200 );
    guiDrawLine( x, y - 3, tx, ty - 3 );
    guiSetColor( 200, 200, 200 );
    guiDrawSquare( x, y, 10 );
  };
};
#-----------------------------------------------------------------
# 建物
Building = Char + object {
  init = method( _x ) {
    init_char( _x, HEIGHT );
    diameter = irand(50) + 50;
    throw building;
  };
  update = method() { };
  draw = method() {
    i = 0;
    while ( i < 10 ) {
      d = 100 + i * 10;
      guiSetColor( d, d, d );
      guiDrawCircle( x, y, diameter * (10 - i) / 10 );
      i = i + 1;
    }
  };
  destroy = method() {
    ObjectList.remove( self );
    e = new BuildingExplosion;
    e.init( x, y );
    dispose building;
  };
};
#-----------------------------------------------------------------
# 建物の爆発
BuildingExplosion = Char + object {
  timer = 30;
  init = method( _x, _y ) {
    init_char( _x, _y );
  };
  update = method() {
    timer = timer - 1;
    if ( timer < 0 ) {
      ObjectList.remove( self );
```

```
    }
  };
  draw = method() {
    i = 0;
    while ( i < 2 ) {
      cr = irand(50) + 200;
      cg = irand(50) + 200;
      cb = irand(50) + 200;
      r = irand(10) + 50;
      px = x + irand(60) - 30;
      py = y - irand(40);
      guiSetColor( cr, cg, cb );
      guiDrawCircle( px, py, r );
      i = i + 1;
    }
  };
};
#---------------------------------------------------------------------
# ハンドラ

# An explosion destroys a balloon.
join *e.explosion() bln.balloon(child) where e.is_collided( bln ) {
  bln.destroy();
  child.set_vel( 0, 2 );
};

# An explosion destroys a bomb.
join *e.explosion() b.bomb() where e.is_collided( b ) {
  b.destroy();
  explosion = new Explosion;
  explosion.init( b.x, b.y, 1.2 );
};

# A missile destroys a balloon.
join m.missile() bln.balloon(child) where m.is_collided( bln ) {
  m.destroy();
  bln.destroy();
  child.set_vel( 0, 2 );  # drop a child bomb
};

# A missile destroys a bomb.
join m.missile() b.bomb() where m.is_collided( b ) {
  m.destroy();
  b.destroy();
};

# A balloon holds its child bomb at the tip of its string.
join *bln.balloon(child) *b.bomb() where child == b {
  b.set_pos( bln.tipx, bln.tipy );
};

# A bomb destroys a building.
join b.bomb() bld.building() where b.is_collided(bld) {
  bld.destroy();
  b.destroy();
};
```

```
# An explosion destroys a building.
# One explosion cannot destroy 2 or more buildings.
join e.explosion() bld.building() where e.is_collided(bld) {
  bld.destroy();
};


#---------------------------------------------------------------
# main

# 砲台生成
Battery.init();

# 建物生成
i = 0;
while ( i < 6 ) {
  b = new Building;
  b.init( i * 70 + 120 );
  i = i + 1;
}

# メインループ
init_game = method() {
  guiSetClearColor( 50, 50, 120 );
};
update_game = method() {
  BalloonGenerator.update();
  ObjectList.iterate( method( e ) { e.update(); } );
  ignition;
};
draw_game = method() {
  ObjectList.reverse_iterate( method( e ) { e.draw(); } );
  guiSetColor( 200, 100, 100 );
  guiSetLineWidth( 4 );
  guiDrawLine( 0, HEIGHT, WIDTH, HEIGHT );
};
game_loop( WIDTH, HEIGHT, init_game, update_game, draw_game, 10 );
```

B.4 アクションゲーム Descender の Mogemoge プログラム

4 章で説明した，3 つのゲームのうち，Descender の全ソースコードを次に掲載する．

```
WIDTH  = 512;
HEIGHT = 512;

PLAYER_Y           = HEIGHT * 3 / 5;
BUILDING_SIZE      = 80;
WINDOW_SIZE        = 50;
WINDOW_STEP        = 80;
LEFT_PLAYER_X      = BUILDING_SIZE + 20;
RIGHT_PLAYER_X     = WIDTH - BUILDING_SIZE - 20;
SCROLL_OUT_LIMIT_Y = -40;
BREADTH_TO_CHANGE_ROPE = 8;

#---------------------------------------------------------------------
# プレーヤーの描画
PlayerDrawer = object {
  left_arm   = 0;
  right_arm  = 0;
  left_leg   = 0;
  right_leg  = 0;
  anim_timer = 0;
  draw_arm = method( x, y, type, dir ) {
    r   = 10;
    hy  = y + r;
    len = 30;

    guiSetLineWidth( 3 );
    guiSetColor( 255, 200, 80 );

    if ( type == 0 ) {
      guiDrawLine( x + r * dir, hy, x + 4 * dir, hy - len );

    } elif ( type == 1 ) {
      x0 = r;      y0 = 0;
      x1 = r + 5;  y1 = -len / 4;
      x2 = 4 * dir;y2 = -len / 2;
      x0 = x0 * dir;
      x1 = x1 * dir;
      guiDrawLine( x + x0, hy + y0, x + x1, hy + y1 );
      guiDrawLine( x + x1, hy + y1, x + x2, hy + y2 );

    } elif ( type == 2 ) {
      d = r * dir;
      guiDrawLine( x + d, hy, x + d + 20 * dir, y - 3 );

    } elif ( type == 3 ) {
      d = r * dir;
      guiDrawLine( x + d, hy, x - d + 25 * dir, y - 3 );
    }
```

```
};
draw_leg = method( x, y, type, dir ) {
  r    = 20;
  r2   = r / 2;
  ly   = y + r + r2;
  llen = 20;
  guiSetLineWidth( 3 );
  if ( type == 0 ) {
    guiDrawLine( x + r2 * dir, ly, x + 3 * dir, ly + llen );

  } elif ( type == 1 ) {
    x0 = r2;            y0 = 0;
    x1 = r2 + llen / 4; y1 = llen / 4;
    x2 = 3 * dir;       y2 = llen / 2;
    x0 = x0 * dir;
    x1 = x1 * dir;
    guiDrawLine( x + x0, ly + y0, x + x1, ly + y1 );
    guiDrawLine( x + x1, ly + y1, x + x2, ly + y2 );

  } elif ( type == 2 ) {
    d = r2 * dir;
    guiDrawLine( x + d, ly, x + d + dir * 8, ly + llen );

  } elif ( type == 3 ) {
    d = r2 * dir;
    guiDrawLine( x + d, ly, x + d - dir * 4, ly + llen );
  }
};
draw_human = method( x, y ) {
  draw_arm( x, y, left_arm, -1 );
  draw_arm( x, y, right_arm, 1 );

  draw_leg( x, y, left_leg, -1 );
  draw_leg( x, y, right_leg, 1 );

  r = 20;
  guiSetColor( 255, 255, 255 );
  guiDrawCircle( x, y, r );
  guiDrawSquare( x, y + r, r );
  guiSetLineWidth( 3 );
};
progress = method() {
  anim_timer = anim_timer + 1;
};
update_descending = method() {
  n = anim_timer / 3;
  left_arm  = right_leg = n % 2;
  right_arm = left_leg  = (n + 1) % 2;
};
update_horizontal_move = method( ) {
  n = anim_timer / 2;
  left_arm  = left_leg  = n % 2 + 2;
  right_arm = right_leg = ( n + 1 ) % 2 + 2;
};
set_right_arm = method( type ) {
  right_arm = type;
```

```
  };
  set_left_arm = method( type ) {
    left_arm = type;
  };
  set_right_leg = method( type ) {
    right_leg = type;
  };
  set_left_leg = method( type ) {
    left_leg = type;
  };
};
#-----------------------------------------------------------------
# 描画のためのベースクラス
Drawer = object {
  draw_bird = method( x, y, dir, flap ) {
    guiSetColor( 30, 200, 30 );
    guiDrawCircle( x, y, 30 );
    guiDrawCircle( x + dir * 15, y - 8, 15 );
    ofsy = 0;
    if ( flap ) {
      ofsy = -20;
    } else {
      ofsy = 0;
    }
    guiSetColor( 30, 150, 30 );
    guiDrawCircle( x - dir * 4, y + ofsy, 20 );
  };

  draw_bitch = method( x, y ) {
    guiSetColor( 0, 0, 0 );
    guiDrawCircle( x, y, 12 );
    guiSetColor( 200, 80, 80 );
    guiDrawCircle( x + 2, y + 2, 12 );
  };

  draw_vertical_rope = method( x, top_y, bottom_y, flying ) {
    guiSetLineWidth( 3 );
    if ( flying ) {
      guiSetColor( irand(100)+100, 140, 160 );
    } else {
      guiSetColor( 140, 140, 160 );
    }
    guiDrawLine( x, top_y, x, bottom_y );
  };

  draw_holizontal_rope = method( x, y, tx, flying ) {
    guiSetLineWidth( 3 );
    if ( flying ) {
      guiSetColor( irand(100) + 100, 140, 160 );
    } else {
      guiSetColor( 140, 140, 160 );
    }
    x0 = x + (tx - x) * step;
    guiDrawLine( x, y, x0, y );
    if ( step < 1 ) {
      guiSetColor( 200, 200, 200 );
```

```
    r = 0;
    if ( x > tx ) {
      r = PI;
    }
    guiDrawArrowhead( x0, y, r, 5 );
  }
};

draw_window = method( x, y ) {
  guiSetColor( 0, 80, 255 );
  guiDrawSquare( x, y, WINDOW_SIZE );
  guiSetColor( 64, 64, 64 );
  guiSetLineWidth( 2 );
  x0 = x - WINDOW_SIZE / 2;
  y0 = y - WINDOW_SIZE / 2;
  guiDrawLine( x0, y0, x0 + WINDOW_SIZE, y0 );
  guiDrawLine( x0, y0, x0, y0 + WINDOW_SIZE );
};

draw_inhabitant = method( x, y, w, h, ofsy ) {
  py = y + ofsy;
  guiSetClip( x, y, w, h );
  guiSetColor( 255, 0, 0 );
  cx = x + w / 2;
  cy = py + 20;
  guiDrawCircle( cx, cy, 30 );
  guiFillRect( cx - 10, cy + 15, 20, 20 );
  guiSetColor( 0, 0, 0 );
  guiSetLineWidth( 3 );
  guiDrawLine( cx - 8, cy - 8, cx - 4, cy - 4 );
  guiDrawLine( cx + 8, cy - 8, cx + 4, cy - 4 );
  guiDrawLine( cx - 4, cy + 4, cx, cy + 6 );
  guiDrawLine( cx + 4, cy + 4, cx, cy + 6 );
  guiClearClip( );
};

draw_arm = method( x, y, to_x, w ) {
  ofsx = ( to_x - x ) * w;
  guiSetColor( 255, 0, 0 );
  guiSetLineWidth( 8 );
  guiDrawLine( x, y, x + ofsx, y );
};

draw_explosion = method( x, y, timer, diameter ) {
  guiSetColor( 255, irand(255), 0 );
  guiDrawCircle( x, y, diameter * timer / 20 );
};

draw_buildings = method( ) {
  guiSetColor( 200, 200, 200 );
  guiFillRect( 0, 0, BUILDING_SIZE, HEIGHT );
  guiFillRect( WIDTH - BUILDING_SIZE, 0, WIDTH, HEIGHT );
};

draw_cloud = method( x, y ) {
  guiSetColor( 180, 180, 220 );
```

```
    guiDrawCircle( x, y, 30 );
    guiDrawCircle( x + 10, y + 5, 30 );
    guiDrawCircle( x - 15, y + 8, 30 );
    guiDrawCircle( x - 40, y + 8, 50 );
  };
};
#-----------------------------------------------------------------
# プレーヤー
Player = object {
  x       = RIGHT_PLAYER_X;
  y       = PLAYER_Y;
  drawer  = new PlayerDrawer;
  throw player;

  update_descending = method() {
    if ( guiKeyPressed( KEY_DOWN ) ) {
      throw cmd_descend;
    }
    if ( guiKeyOn( KEY_LEFT ) ) {
      throw cmd_shoot_hrope;
    } elif ( guiKeyOn( KEY_RIGHT ) ) {
      throw cmd_shoot_hrope;
    }
    is_left_side = ( x == LEFT_PLAYER_X );
    if (guiKeyPressed( KEY_RIGHT ) and is_left_side) {
      throw cmd_go_side;
    } elif (guiKeyPressed( KEY_LEFT ) and not is_left_side) {
      throw cmd_go_side;
    }
  };
  update_moving_horizontally = method() {
    dx = 0;
    if ( guiKeyPressed( KEY_LEFT ) ) {
      dx = -4;
    } elif ( guiKeyPressed( KEY_RIGHT ) ) {
      dx = 4;
    } elif ( guiKeyOn( KEY_DOWN ) ) {
      throw cmd_go_down;
    }

    x = x + dx;
    if ( x > RIGHT_PLAYER_X ) {
      x = RIGHT_PLAYER_X;
    } elif ( x < LEFT_PLAYER_X ) {
      x = LEFT_PLAYER_X;
    } elif ( dx != 0 ) {
      drawer.progress();
    }

    drawer.update_horizontal_move();
  };
  update_falling = method() {
    y = y + 5;
    drawer.set_left_arm( 0 );
    drawer.set_right_arm( 0 );
    drawer.set_left_leg( 2 );
```

```
    drawer.set_right_leg( 2 );
    if ( y > HEIGHT + 40 ) {
      ObjectList.remove( self );
    }
  };
  update = update_descending;

  draw = method() {
    drawer.draw_human( x, y );
  };

  is_collided = method( obj ) {
    dx = x - obj.x;
    dy = y - obj.y;
    r = 26;
    result ( dx * dx + dy * dy ) < r * r;
  };

  anim_shoot_hrope = method( is_leftside ) {
    if ( is_leftside ) {
      drawer.set_left_arm( 2 );
    } else {
      drawer.set_right_arm( 2 );
    }
  };

  anim_descend = method() {
    drawer.progress();
    drawer.update_descending();
  };
};
#-----------------------------------------------------------------
# 垂直ロープ
VerticalRope = Drawer + object {
  x  = 0;
  y  = 0;
  by = HEIGHT;
  ey = -1;
  init = method( _x, _y ) {
    x = _x;
    y = _y;
    ObjectList.add( self );
    throw v_rope;
  };
  init_falling = method( _x, _y, _by ) {
    x  = _x;
    y  = _y;
    by = _by;
    ObjectList.add( self );
    update = update_falling;
  };
  destroy = method() {
    dispose v_rope;
    ObjectList.remove( self );
  };
  update = method() {};
```

```
  update_falling = method() {
    y = y + 10;
    by = min( HEIGHT, by + 10 );
    if ( y > HEIGHT ) {
      ObjectList.remove( self );
    }
  };
  update_extending = method() {
    ey = min( HEIGHT, ey + 20 );
    if ( ey == HEIGHT ) {
      update = method() {};
      by = ey;
    }
  };
  draw = method() {
    draw_vertical_rope( x, y, by, false );
    if ( update == update_extending ) {
      draw_vertical_rope( x, by, ey, true );
    }
  };
  is_on = method( _x, _y ) {
    result x == _x and y <= _y and _y <= by;
  };
  extend_to_bottom = method() {
    if ( ey < 0 ) {
      ey = by;
      update = update_extending;
    }
  };
  cut = method( cut_y ) {
    fr = new VerticalRope;
    fr.init_falling( x, cut_y, by );
    throw cut_vrope( by, cut_y );
    by = cut_y;
  };
};
#---------------------------------------------------------------------
# 水平ロープ
HorizontalRope = Drawer + object {
  x = tx = y = step = 0;
  init = method( _y, _is_left ) {
    y = _y;
    if ( _is_left ) {
      x = WIDTH - BUILDING_SIZE;
      tx = BUILDING_SIZE;
    } else {
      x = BUILDING_SIZE;
      tx = WIDTH - BUILDING_SIZE;
    }
    step = 0;
    ObjectList.add( self );
    throw bg_object;
    throw h_rope_flying;
  };
  destroy = method() {
    dispose bg_object;
```

```
    dispose h_rope;
    ObjectList.remove( self );
  };
  update_flying = method() {
    step = step + 0.05;
    if ( step >= 1 ) {
      step = 1;
      throw h_rope;
      dispose h_rope_flying;
      update = update_ok;
    }
  };
  update_ok = method() {
  };
  update = update_flying;
  draw = method() {
    b = ( update == update_flying );
    draw_holizontal_rope( x, y, tx, b );
  };
};
#-----------------------------------------------------------------
# 鳥
Bird = Drawer + object {
  x          = 0;
  y          = 0;
  ry         = 0;
  dir        = 1;
  vel        = 0;
  timer      = irand( 20 ) + 30;
  anim_timer = 0;
  init = method( ) {
    dir = irand( 2 ) * 2 - 1;
    if ( dir < 0 ) {
      x = WIDTH;
    } else {
      x = 0;
    }
    y = HEIGHT / 2 - irand( 100 ) - 50;
    vel = dir * ( irand( 4 ) + 2 );
    ObjectList.add( self );
    throw bg_object;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bg_object;
  };
  update = method() {
    ry = ry + PI / 32;
    y = y + sin( ry ) * 3;
    x = x + vel;
    if ( x < 0 or x > WIDTH ) {
      destroy();
    }
    timer = timer - 1;
    if ( timer < 0 ) {
      timer = irand( 40 ) + 5;
```

```
      b = new Bitch;
      b.init( x, y );
    }
    anim_timer = anim_timer + 1;
  };
  draw = method() {
    flap = ( anim_timer / 2 % 2 == 0 );
    draw_bird( x, y, dir, flap );
  };
};
#---------------------------------------------------------------------
# 鳥の爆弾
Bitch = Drawer + object {
  x = y = 0;
  init = method( _x, _y ) {
    x = _x;
    y = _y;
    ObjectList.add( self );
    throw bg_object;
    throw bitch;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bg_object;
    dispose bitch;
  };
  update = method() {
    y = y + 8;
    if ( y > HEIGHT ) {
      destroy();
    }
  };
  draw = method() {
    draw_bitch( x, y );
  };
};
#---------------------------------------------------------------------
# 鳥を生成するプログラム
BirdGenerator = object {
  timer = 300;
  count = 0;
  update = method() {
    timer = timer - 1;
    if ( timer < 0 ) {
      b = new Bird;
      b.init();
      count = count + 1;
      n = min( 50, 500 - count * 30 );
      timer = irand( 200 ) + 100;
    }
  };
};
#---------------------------------------------------------------------
# ビルの窓
Window = Drawer + object {
  x = 0;
```

```
  y = 0;
  init = method( _x, _y ) {
    x = _x;
    y = _y;
    ObjectList.add( self );
    throw bg_object;

    if ( irand( 10 ) < 3 ) {
      s = WINDOW_SIZE / 2;
      i = new Inhabitant;
      i.init( _x - s, _y - s, WINDOW_SIZE, WINDOW_SIZE );
    }
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bg_object;
  };
  update = method() {
  };
  draw = method() {
    draw_window( x, y );
  };
};
#-----------------------------------------------------------------
# ビルの住人
Inhabitant = Drawer + object {
  x = y = w = h = 0;
  ofsy = 0;
  timer = 0;
  init = method( _x, _y, _w, _h ) {
    x = _x;
    y = _y;
    w = _w;
    h = _h;
    ObjectList.add( self );
    throw bg_object;

    ofsy = WINDOW_SIZE;
    timer = irand(100) + 50;
    update = update_hiding;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bg_object;
  };
  update = method() {
  };
  update_hiding = method() {
    if ( y < PLAYER_Y + 50 ) {
      timer = timer - 1;
      if ( timer < 0 ) {
        update = update_appearing;
      }
    }
  };
  update_appearing = method() {
```

```
    ofsy = ofsy - 1;
    if ( ofsy <= 0 ) {
      create_arm();
      update = method() {};
    }
  };
  create_arm = method() {
    arm = new Arm;
    cx = x + w / 2;
    if ( x < WIDTH / 2 ) {
      arm.init( cx, y + 40, cx + 60 );
    } else {
      arm.init( cx, y + 40, cx - 60 );
    }
  };
  draw = method() {
    draw_inhabitant( x, y, w, h, ofsy );
  };
};
#----------------------------------------------------------------
# 住人の腕
Arm = Drawer + object {
  x = to_x = y = timer = 0;
  init = method( _x, _y, _to_x ) {
    x    = _x;
    to_x = _to_x;
    y    = _y;
    ObjectList.add( self );
    throw bg_object;
  };
  destroy = method() {
    ObjectList.remove( self );
    dispose bg_object;
  };
  update_extending = method() {
    if ( timer < 15 ) {
      timer = timer + 1;
    } else {
      throw cmd_cut_rope;
      update = update_shrinking;
    }
  };
  update_shrinking = method() {
    if ( timer > 0 ) {
      timer = timer - 1;
    } else {
      destroy();
    }
  };
  update = update_extending;
  draw = method() {
    w = timer / 15.0;
    draw_arm( x, y, to_x, w );
  };

  can_cut = method( rope ) {
```

```
    b = rope.y < y and y < rope.by
        and min( x, to_x ) <= rope.x and rope.x <= max( x, to_x );
    result b;
  };
};
#-----------------------------------------------------------------
# ビルの窓の生成
WindowGenerator = object {
  y      = 0;
  last_y = 0;
  throw bg_object;
  destroy = method() {};
  update = method() {
    if ( y < last_y ) {
      cy = HEIGHT + WINDOW_SIZE / 2;
      w = new Window;
      w.init( BUILDING_SIZE / 2, cy );
      w = new Window;
      w.init( WIDTH - BUILDING_SIZE / 2, cy );
      last_y = y - WINDOW_STEP;
    }
  };
};
#-----------------------------------------------------------------
# プレーヤーに鳥の爆弾が当たった時の効果
Explosion = Drawer + object {
  x = y = timer = diameter = 0;
  init = method( _x, _y, _diameter ) {
    x = _x;
    y = _y;
    diameter = _diameter;
    timer = 20;
    ObjectList.add( self );
  };
  destroy = method() {
    ObjectList.remove( self );
  };
  update = method() {
    timer = timer - 1;
    if ( timer < 0 ) {
      destroy();
    }
  };
  draw = method() {
    draw_explosion( x, y, timer, diameter );
  };
};
#-----------------------------------------------------------------
# 背景の雲
Cloud = Drawer + object {
  x = y = dir = 0;
  init = method() {
    x = irand( WIDTH );
    y = irand( HEIGHT );
    dir = ( irand( 2 ) * 2 - 1 ) * ( rand() * 2 + 0.5);
    ObjectList.add( self );
```

```
    throw cloud;
  };
  update = method() {
    x = x + dir;
    if ( x < 0 ) {
      x = WIDTH;
    } elif ( x > WIDTH ) {
      x = 0;
    }
    if ( y < -40 ) {
      y = HEIGHT + 40;
    }
  };
  draw = method() {
    draw_cloud( x, y );
  };
};
#---------------------------------------------------------------------
# ビル
Buildings = object {
  update = method() {};
  draw = method() {
    Drawer.draw_buildings();
  };
};
#---------------------------------------------------------------------
# 垂直ロープで降りるコマンド
join p.cmd_descend *r.v_rope where r.is_on( p.x, p.y ) {
  d = min( r.by - p.y, 2 );
  if ( d > 0 ) {
    throw scroll( d );
    p.anim_descend();
  } else {
    r.extend_to_bottom();
  }
};
join p.cmd_descend { };
#---------------------------------------------------------------------
# 水平移動コマンド処理

join p.cmd_go_side *r.h_rope
        where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
  p.update = p.update_moving_horizontally;
  throw scroll( r.y - p.y );
};
join p.cmd_go_side { };
#---------------------------------------------------------------------
# 水平ロープを掴んでいるときに，下へ降りる/垂直ロープを張る処理
join p.cmd_go_down *r.v_rope
    where abs( p.x - r.x ) < BREADTH_TO_CHANGE_ROPE
      and r.y <= p.y and p.y <= r.by {
  p.update  = p.update_descending;
  p.x       = r.x;
};
join p.cmd_go_down
      where abs( p.x - LEFT_PLAYER_X ) < BREADTH_TO_CHANGE_ROPE {
```

```
  r = new VerticalRope;
  r.init( LEFT_PLAYER_X, p.y );
  r.by = p.y;
  r.extend_to_bottom();
};
join p.cmd_go_down
      where abs( p.x - RIGHT_PLAYER_X ) < BREADTH_TO_CHANGE_ROPE {
  r = new VerticalRope;
  r.init( RIGHT_PLAYER_X, p.y );
  r.by = p.y;
  r.extend_to_bottom();
};
join p.cmd_go_down { };
#---------------------------------------------------------------------
# 画面スクロール処理
join *any.scroll(d) *o.bg_object {
  o.y = o.y - d;
  if ( o.y < SCROLL_OUT_LIMIT_Y ) {
    o.destroy();
  }
};
join *any.scroll(d) *r.v_rope {
  if ( r.y > 0 or d > 0 ) {
    r.y = r.y - d;
    if ( r.y < 0 ) {
      r.y = 0;
    }
  }
  if ( r.by < HEIGHT ) {
    r.by = r.by - d;
    if ( r.by < 0 ) {
      r.destroy();
    }
  }
};
join *any.scroll(d) *c.cloud {
  c.y = c.y - d / 2.0;
};
join any.scroll { };
#---------------------------------------------------------------------
# 水平ロープを張る処理
join p.cmd_shoot_hrope *r.h_rope_flying
    where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope *r.h_rope
    where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope {
  hr = new HorizontalRope;
  if ( p.x == LEFT_PLAYER_X ) {
    hr.init( p.y, false );
    p.anim_shoot_hrope( false );
  } else {
    hr.init( p.y, true );
    p.anim_shoot_hrope( true );
  }
```

```
};
join p.cmd_shoot_hrope { };
#---------------------------------------------------------------------
# プレーヤーと鳥の爆弾衝突処理
join *p.player o.bitch where p.is_collided( o ) {
  p.update = p.update_falling;
  e = new Explosion;
  e.init( o.x, o.y, 40 );
  o.destroy();
};
#---------------------------------------------------------------------
# ビルの住人の腕が垂直ロープを切る処理
join a.cmd_cut_rope *r.v_rope where a.can_cut( r ) {
  r.cut( a.y );
};
join a.cmd_cut_rope { };
#---------------------------------------------------------------------
# プレーヤーの掴んでいる垂直ロープが住人の腕に切られた処理
join *p.player rope.cut_vrope( by, cut_y )
    where p.x == rope.x
      and rope.y <= p.y and p.y <= by
      and cut_y < p.y {
  p.update = p.update_falling;
};
join any.cut_vrope { };
#---------------------------------------------------------------------
# main

# 雲生成
i = 0;
while ( i < 4 ) {
  c = new Cloud;
  c.init();
  i = i + 1;
};

# ビル生成
ObjectList.add( Buildings );

# 最初の垂直ロープ 2 本を生成
lr = new VerticalRope;
lr.init( LEFT_PLAYER_X, 0 );
rr = new VerticalRope;
rr.init( RIGHT_PLAYER_X, 0 );

# メインループ
init_game = method() {
  guiSetClearColor( 80, 80, 120 );
};
update_game = method() {
  WindowGenerator.update();
  BirdGenerator.update();
  ObjectList.iterate( method( e ) { e.update(); } );
  Player.update();
  ignition;
};
```

```
draw_game = method() {
  ObjectList.reverse_iterate( method( e ) { e.draw(); } );
  Player.draw();
};
game_loop( WIDTH, HEIGHT, init_game, update_game, draw_game, 40 );
```

B.5 シューティングゲーム Balloon の Ruby プログラム

4.5.5 節で説明した，Balloon の Ruby による全ソースコードを次に掲載する．

```
require "tk"
require 'singleton'

include Math

WIDTH  = 512
HEIGHT = 512
G      = 4.0 / 30
$frame_timer = 0

#--------------------------------------------------------------------
# Tk 初期化
frame = TkFrame.new(nil, 'relief'=>'sunken', 'borderwidth'=>4)
frame.pack
$canvas = TkCanvas.new(frame,
                       'width'=>WIDTH,
                       'height'=>HEIGHT,
                       'background'=>'darkgray')
$canvas.pack('fill'=>'both')

# Tk 入力処理
$key_stat = { }
$key_pre  = { }
$key_on   = { }

# マウスボタン binding
$canvas.bind('Button-1', Proc.new{$key_stat[:button1]=true})
$canvas.bind('Button-2', Proc.new{$key_stat[:button2]=true})
$canvas.bind('Button-3', Proc.new{$key_stat[:button3]=true})
$canvas.bind('ButtonRelease-1', Proc.new{$key_stat[:button1]=nil})
$canvas.bind('ButtonRelease-2', Proc.new{$key_stat[:button2]=nil})
$canvas.bind('ButtonRelease-3', Proc.new{$key_stat[:button3]=nil})

# キーボード入力 binding
Tk.root.bind('KeyPress',
               proc{|e| $key_stat[e.keysym.to_sym]=true })
Tk.root.bind('KeyRelease',
               proc{|e| $key_stat[e.keysym.to_sym]=nil })

#--------------------------------------------------------------------
# キー入力処理
def update_key
  $key_on = {}
  $key_stat.keys.each do |k|
    $key_on[k] = true if $key_stat[k] && !$key_pre[k]
  end
  $key_pre = $key_stat.dup
end
```

```
#---------------------------------------------------------------------
# ゲームオブジェクトの配列の操作
$obj_list = []
def $obj_list.add( o )
  push o if !o.nil? && !include?( o )
end
def $obj_list.remove( o )
  delete_if { |e| e == o }
end

# 色コード変換
def to_color( r, g, b )
  [r, g, b].inject('#') { |r,n| r + ("%2.2x" % n) }
end

#---------------------------------------------------------------------
# キャラクタのベースになるクラス
class Char
  attr_accessor :x, :y, :vx, :vy, :spd, :diameter, :widget
  def initialize( _x, _y )
    set_pos( _x, _y )
    set_vel( 0, 0 )
    @diameter = 16
    @widget = []
    $obj_list.add self
  end
  def destroy
    destroy_shape
    $obj_list.remove self
  end
  def set_pos( _x, _y )
    @x = _x
    @y = _y
  end
  def set_vel( _vx, _vy )
    @vx = _vx
    @vy = _vy
  end
  def update_position
    @x += @vx
    @y += @vy
  end
  def is_collided( obj )
    dx = obj.x - @x
    dy = obj.y - @y
    r = (@diameter + obj.diameter) / 2
    return dx * dx + dy * dy < r * r
  end
  def update_shape
    if @widget.size == 0
      shape_classes().zip( shape_coords(), shape_options() ).each do |z|
        w = z[0].new( $canvas, *z[1] )
        @widget.push w
        $canvas.itemconfigure( w.id, *z[2] ) if z[2].size > 0
      end
```

```
    end
    @widget.zip( shape_coords(), shape_options() ).each do |z|
      $canvas.coords( z[0].id, *z[1] )
      $canvas.itemconfigure( z[0].id, *z[2] )
    end
  end
  def destroy_shape
    @widget.each do |w|
      $canvas.delete w
    end
  end
  def shape_classes
    [ TkcOval ]
  end
  def shape_options
    [ [ 'outline'=>'coral4', 'fill'=>'coral' ] ]
  end
  def shape_coords
    r = @diameter / 2
    [ [ @x - r, @y - r, @x + r, @y + r ] ]
  end
end
#--------------------------------------------------------------------
# 風船
class Balloon < Char
  attr_accessor :tipx, :tipy, :bomb
  def initialize( x )
    super( x, 0 )
    @swing_cycle = 0
    @swing_angle = 0
    @string_len  = 50
    @tipx        = 0
    @tipy        = 0
    set_vel( 0, rand() * 1.2 + 0.3 );
    @r = rand(128) + 100;
    @g = rand(128) + 100;
    @b = rand(128) + 100;
    @bomb = Bomb.new( x, 0, self );
  end
  def update
    @swing_cycle += PI / 40;
    update_position
    update_tip_pos
    if !@bomb.nil?
      @bomb.set_pos( @tipx, @tipy )
    end
    destroy() if ( @y > HEIGHT )
  end
  def update_tip_pos
    @swing_angle = sin( @swing_cycle ) * PI / 4;
    @tipx = @x + sin( @swing_angle ) * @string_len;
    @tipy = @y + 8 + cos( @swing_angle ) * @string_len;
  end
  def shape_classes
    [ TkcOval, TkcLine ]
  end
```

```
  def shape_options
    [
      [ 'outline'=>'coral4', 'fill'=>'coral' ],
      [ 'fill'=>'white' ],
    ]
  end
  def shape_coords
    r = @diameter / 2
    [
      [ @x - r, @y - r, @x + r, @y + r ],
      [ @x, @y + 8, @tipx, @tipy ]
    ]
  end
end
#--------------------------------------------------------------------
# 爆弾
class Bomb < Char
  attr_accessor :parent
  def initialize( _x, _y, _parent )
    super( _x, _y )
    @diameter = 6;
    @parent = _parent;
  end
  def destroy
    destroy_shape
    $obj_list.remove self
    if !@parent.nil?
      @parent.bomb = nil
    end
  end
  def update
    update_position
    if ( y > HEIGHT )
      destroy();
      Explosion.new( x, y, 1 )
    end
  end
  def shape_classes
    [ TkcRectangle ]
  end
  def shape_options
    [ [ 'fill'=>'red' ], ]
  end
end
#--------------------------------------------------------------------
# 風船の生成
class BalloonGenerator
  include Singleton
  def initialize
    @timer = 0;
  end
  def update
    n = 200 - $frame_timer / 20;
    if ( n < 50 )
      n = 50;
    end
```

```
    if ( $frame_timer % n == 0 )
      Balloon.new( rand() * WIDTH );
    end
  end
end
#--------------------------------------------------------------------
# ミサイル
class Missile < Char
  def initialize( _x, _y, _vx, _vy )
    super( _x, _y )
    @timer = 0
    set_vel( _vx, _vy );
    @diameter = 10;
  end
  def update
    @vy += G
    @timer += 1
    if @timer > 10 && !$key_stat[:space]
      destroy()
      Explosion.new( @x, @y, 0.8 )
    end
    update_position
    if @x > WIDTH || @y > HEIGHT
      destroy();
    end
  end
  def shape_options
    [ [ 'outline'=>'black', 'fill'=>'blue' ] ]
  end
end
#--------------------------------------------------------------------
# 爆発
class Explosion < Char
  attr_accessor :active
  def initialize( _x, _y, _diameter_scale )
    super( _x, _y )
    @diameter = 100 * _diameter_scale
    @timer = 0
    @active = true
  end
  def update
    @timer += 1;
    if @timer > 15
      destroy();
    end
  end
  def shape_options
    n = ( rand(2) % 2 ) * 100;
    [ [ 'fill'=>to_color( 180, n, n ) ] ]
  end
end
#--------------------------------------------------------------------
# 砲台
class Battery < Char
  def initialize
    super( 0, HEIGHT )
```

```
    @dir = -PI / 4;
    @tx  = 0;
    @ty  = 0;
    @dx  = 0;
    @dy  = 0;
    @len = 40;
  end

  def update
    ulim = -PI / 2;
    blim = 0;

    if $key_stat[:Up]
      @dir = @dir - PI / 40
      if @dir < ulim
        @dir = ulim
      end
    elsif $key_stat[:Down]
      @dir = @dir + PI / 40
      if @dir > blim
        @dir = blim
      end
    end

    @dx = cos( @dir );
    @dy = sin( @dir );
    @tx = @x + cos( @dir ) * @len;
    @ty = @y + sin( @dir ) * @len;

    if $key_on[:space]
      spd = 10;
      missile = Missile.new( @tx, @ty, @dx * spd, @dy * spd )
    end
  end
  def shape_classes
    [ TkcLine ]
  end
  def shape_options
    [ [ 'fill'=>'white' ] ]
  end
  def shape_coords
    [ [ @x, @y, @tx, @ty ] ]
  end
end
#---------------------------------------------------------------------
# 建物
class Building < Char
  def initialize( _x )
    super( _x, HEIGHT )
    @diameter = rand(50) + 50
  end
  def update
  end
  def destroy
    destroy_shape
    $obj_list.remove self
```

```
    BuildingExplosion.new( x, y )
  end
  def shape_classes
    [ TkcOval ] * 10
  end
  def shape_options
    (0...10).map do |i|
      d = 100 + i * 10
      c = to_color( d, d, d )
      [ 'fill'=>c, 'outline'=>c ]
    end
  end
  def shape_coords
    (0...10).map do |i|
      r = @diameter * ( 10 - i ) / 10 / 2
      [ @x - r, @y - r, @x + r, @y + r ]
    end
  end
end
#---------------------------------------------------------------------
# 建物の爆発
class BuildingExplosion < Char
  def initialize( _x, _y )
    super( _x, _y )
    @timer = 30
  end
  def update
    @timer -= 1
    if @timer < 0
      destroy
    end
  end
  def shape_classes
    [ TkcOval ] * 2
  end
  def shape_options
    (0...2).map do |i|
      cr = rand(50) + 200
      cg = rand(50) + 200
      cb = rand(50) + 200
      c = to_color( cr, cg, cb )
      [ 'fill'=>c, 'outline'=>c ]
    end
  end
  def shape_coords
    (0...2).map do |i|
      px = x + rand(60) - 30
      py = y - rand(40)
      r = rand(10) + 50
      [ px - r, py - r, px + r, py + r ]
    end
  end
end
#---------------------------------------------------------------------
# 衝突検査処理下請け
def check_collision( c1, c2 )
```

```
  o1s = $obj_list.find_all { |o| o.kind_of? c1 }
  o2s = $obj_list.find_all { |o| o.kind_of? c2 }
  o1s.each do |o1|
    o2s.each do |o2|
      yield o1, o2 if o1 != o2 && o1.is_collided( o2 )
    end
  end
end
#--------------------------------------------------------------------
# 衝突検査処理
def check_collision_all
  check_collision( Missile, Balloon ) do |m,bln|
    m.destroy; bln.destroy
    bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
  end
  check_collision( Missile, Bomb ) do |m,b|
    m.destroy; b.destroy
  end
  check_collision( Explosion, Balloon ) do |e,bln|
    bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
    bln.destroy
  end
  check_collision( Explosion, Bomb ) do |e,b|
    b.destroy()
    Explosion.new( b.x, b.y, 1.2 )
  end
  check_collision( Building, Bomb ) do |bld,b|
    bld.destroy; b.destroy
  end
  check_collision( Explosion, Building ) do |e,bld|
    if e.active then e.active = false; bld.destroy end
  end
end

#--------------------------------------------------------------------
# main

# 砲台生成
b = Battery.new
# 建物生成
(0..6).each do |i|
  b = Building.new( i * 70 + 120 )
end

# メインループ
after = TkAfter.new
after.set_start_proc(20, proc {
  $frame_timer += 1
  begin
    BalloonGenerator.instance.update
    update_key
    $obj_list.each { |o| o.update }
    check_collision_all
    $obj_list.each { |o| o.update_shape }
    exit if $key_on[:Escape]
    after.restart
```

```
  rescue => e
    e.backtrace.each { |s| p s }
    p e
  end
})
after.restart
Tk.mainloop
```