情報処理学会論文誌　プログラミング　Vol. 1　No. 2　42–56 (Sep. 2008)

ゲームプログラムに適したリアルタイム性の高いデバッガの提案と実装

丹　野　治　門[†1]

ゲームプログラムのようにインタラクティブ性の高いリアルタイムシステムにおいては，ブレークポイントを用いてプログラムを一時停止させるような従来型のデバッグ手法は適していない．その理由の第 1 は，ゲームプログラムにおいては，プレイヤのキー入力はタイミングや順番が重要であり，これを途中で中断させると，期待どおりの動きをしなくなってしまうためである．理由の第 2 は，ゲーム内のキャラクタの位置座標を確認するなどのたびにプログラムを停止させていては，デバッグの効率が悪くなるためである．以上の問題点を解決するため，本論文ではプログラムの実行位置の情報と変数情報をリアルタイムに更新し，さらにプログラムの実行経路情報をソースコード上にグラデーション表示するデバッガを提案し，ゲームに適した並行処理機構を持つ Java 風オブジェクト指向言語 kameTL およびその統合開発環境上に実装した．本デバッガを用いることにより，ゲーム開発者はプレイを中断せずにデバッグを行うことができるようになるとともに，多数のブレークポイントを設置することなく多くの有用な情報が得られるようになる．本論文では，kameTL で記述されたロールプレイングゲーム，シューティングゲームを例とし，本デバッガの有効性についても述べる．

Design and Implementation of Real-time Debugger for Game Programming

Haruto Tanno[†1]

A typical game program is a highly interactive and real-time system. Thus it is not appropriate to use the frequently-used debugging technique in which the execution of the program is temporally and often suspended by using breakpoints. There are two reasons for this. The first reason is that the timing and order of the keyboard events from the player are very important in playing a game. If the game program is suspended by the debugger, it might not work as expected. The second reason is that the debugging is awfully inefficient if the program is suspended each time the programmer wants to observe the states internal data structures, e.g., coordinates of characters in the game. To solve these problems, this paper proposes a debugger that updates the information of the current execution point and values of variables in real time, and displays the flow of execution by using color gradient within the source code of the program. The proposed debugger is implemented within the integrated development environment of the kameTL, a Java-like object oriented language with concurrent threads designed for game programming. By using this debugger, the programmer are able to debug the game program without frequent suspensions of the play and to obtain much useful information without using many breakpoints. This paper also describes the effectiveness of the debugger through some examples of a roll-playing game and shooting game written in kameTL.

1. はじめに

シューティングゲーム，ロールプレイングゲーム，アクションゲームのようなゲームプログラムは，インタラクティブ性が高いリアルタイムシステムなので，プログラムを一時停止させてしまうような既存のデバッグ手法は向いていない．その理由の第 1 は，プレイヤのキー入力や，多数のキャラクタどうしの相互作用はタイミングや順番が重要であり，これを途中で中断させてしまうと期待どおりの動きをしなくなる可能性があるためである．理由の第 2 は，ソースコード中のどの位置を現在実行しているかといった情報や，ゲーム内のキャラクタの位置座標などの情報を得ようとするたびに，ゲームプレイを中断していては，デバッグの効率が悪くなるためである．

そこで，本研究ではプログラムを停止させず，ゲームプレイを継続しながらデバッグを行える機構を，ゲームシステム記述言語 kameTL [1] およびその統合開発環境上に実装した．kameTL は，ゲームに適した並行処理機構を持つ Java 風のオブジェクト指向言語であり，筆者が開発したものである．以下に本デバッガの特徴を示す．

- ソースコード中の実行位置情報，プログラム内部の変数情報をリアルタイムに表示する（**図 1**）．
- プログラムの実行経路情報をソースコード上にグラデーション表示する（**図 2**）．

結果として，ゲームプログラムの開発者は，ゲームプレイを中断することなくプログラム内部の状態を確認可能になり，多数のブレークポイントを設置することなく多くの情報が得

†1 電気通信大学大学院電気通信学研究科情報工学専攻
Department of Computer Science, The University of Electro-Communications

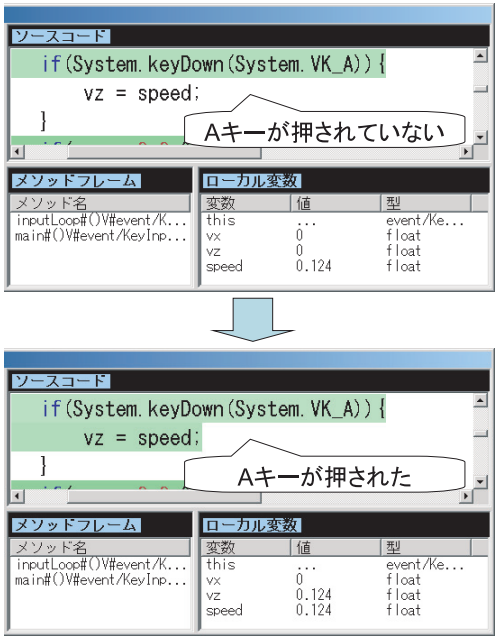

図 1　実行位置情報と変数情報のリアルタイム表示
Fig. 1　Realtime updates of execution point and variable values.

られるようになる．

本論文では，提案するデバッガの設計とその機能，実装の詳細について説明し，実際のゲームプログラムを例にとりその活用事例と有効性についても述べる．

以下，2 章でゲームプログラムを既存手法でデバッグする際の問題点について述べる．3 章で提案するデバッガの特徴および機能を紹介し，4 章で本デバッガの実装の詳細について説明する．5 章では，ロールプレイングゲーム，シューティングゲームを例とし本デバッガの活用事例とその有効性を示し，6 章では，オーバヘッドの計測を行う．7 章では関連研究を紹介し，8 章で本論文のまとめを述べる．

以後，本論文で対象とするゲームは，シューティングゲーム，ロールプレイングゲーム，アクションゲームなどのように，リアルタイム性の高いものに限定することとし，囲碁や将棋のようなものは対象外とする．

```
float vx=0.0,vz=0.0;
float speed = getDeltaTime()*4.0;
if(System.keyDown(System.VK_W)){
    vx = speed;
}
if(System.keyDown(System.VK_Z)){
    vx = -speed;
}
if(System.keyDown(System.VK_S)){
    vz = -speed;
}
if(System.keyDown(System.VK_A)){
    vz = speed;
}
if(vx == 0.0 && vz == 0.0){
    primeActor.setAnimation("stop");
}
else{
    primeActor.setAnimation("walk");
}
primeActor.position.x += vx;
primeActor.position.z += vz;
```

図 2　実行経路情報のグラデーション表示
Fig. 2　Showing execution flow using color gradient.

2. ゲームプログラムのデバッグ

ここでは，ゲームプログラムの特徴と，既存のデバッグ方法の問題点について述べる．

2.1 ゲームプログラム

本論文で対象とするゲームプログラムは一般的には**図 3** のように動作する．まず最初に，画像の読み込みや，三次元モデルデータの読み込みなどの初期化処理が行われる．そして，初期化処理が終わるとループに入り以下の処理を毎秒数十回という時間間隔で繰り返す．この時間間隔を以下，“フレーム”と呼ぶ．

(1)　プレイヤのキー入力を受け取りプレイヤが操作するキャラクタを動かす．
(2)　その他のキャラクタを何らかのルーチンをもとに動かす．
(3)　当たり判定などのキャラクタどうしの相互作用の判定を行う．
(4)　計算の結果を画面に反映させる．

最後に，ゲームが終了すると，プログラムは上のループを抜け，ゲームのセーブデータ保存などの終了処理を行った後に終了する．

ゲームプログラムでは，ループの部分がプログラムの実行時間の大部分を占める．また，

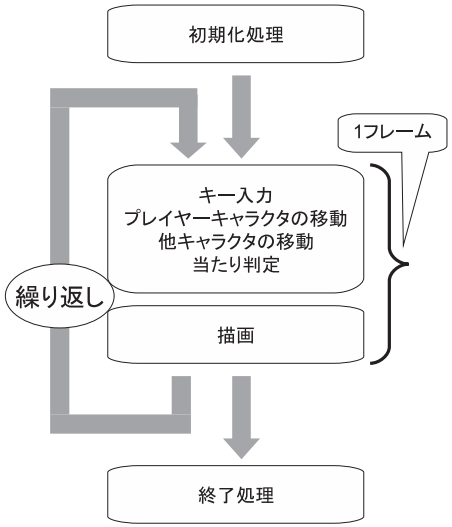

図 3　ゲームプログラムの動き
Fig. 3　Flow of game program.

このループの部分ではプレイヤのキー入力や乱数の値によってプログラム内部の状態が変わっていくため，その挙動は予測しにくい．よって，デバッグに費やされる時間もこのループの部分が最も多い．

2.2　ゲームプログラム中のバグ

ゲームプログラムにおいて，バグの発生する局面とその原因，バグを取り除くためのデバッグ方法は，**表 1** のように分類できる．バグの原因には，論理エラーと実行時エラーの 2 種類がある．

論理エラーは，開発者の設計ミスやコーディングミスが原因で発生し，プログラムが当初の設計とは異なる動きをするものである．一方，実行時エラーは，ゼロによる除算，NULL 値のポインタを参照しようとしたときなどに発生し，その原因は，条件チェック忘れなどのコーディングミスや，プレイヤによる想定外の操作などである．

表 1 (a)，(b) のような，ゲームプログラム中のリアルタイム性の高いループ処理で発生する論理エラーは，既存のデバッガのブレークポイントやステップ実行を用いることでは，取り除くことが難しかったり多大な手間がかかったりする．特にインタラクティブ性が高い

表 1　バグの発生する原因とデバッグ方法
Table 1　Bug and debugging method.

	リアルタイム部分（ループ処理）		非リアルタイム部分（初期化処理，終了処理）
	インタラクティブ	非インタラクティブ	
論理エラー	(a) 既存デバッガではデバッグが難しい	(b) 既存デバッガではデバッグが難しいが，プロファイラが利用できる場合がある	(c) 既存デバッガのステップ実行を用いてプログラム実行時の情報を調べる
実行時エラー	(d) 既存デバッガでプログラム停止時点の情報を調べる		

場合（表 1 (a)）には，プログラムをしばしば一時中断することにより，ゲームプレイ自体が成立しなくなってしまうため，より困難さは大きい．

既存のデバッガの利用以外に，プロファイラを用いて，ソースコード中のどの部分がどれだけ実行されたかを調べてデバッグに利用することも考えられる．たしかに表 1 (b) のように，インタラクティブ性が高くない（非インタラクティブの）場合には，プロファイラによる情報を用いてある程度デバッグすることはできる．しかし，プロファイラで得られるのはあくまでも統計的な情報であるため，表 1 (a) のようにインタラクティブ性の高い場合，プレイヤからの入力やキャラクタどうしの当たり判定などに応じてプログラム内部の状態がどのように変化したかを知ることはできない．

本論文で提案するデバッガは，表 1 (a)，(b) のような，リアルタイム部分の論理エラーを取り除くのに，特にインタラクティブ性が高い場合に有効であることを目指した．

表 1 (c) のような，ゲームプログラム中の初期化処理，終了処理における論理エラーに関しては，既存デバッガを用いてのステップ実行などで，バグの原因を探し出すことができる．また，表 1 (d) のような実行時エラーについては，既存デバッガを用いて，エラー発生時の変数情報を取得するなどして，バグの原因を調べることができる．したがって，表 1 (c)，(d) のようなバグは，提案するデバッガの対象とはしなかった．さらに表 1 に載せなかったものとして，スレッドのデッドロックなどの再現性のないエラーもある．このような再現性のないバグは，既存のデバッグ方式でも除くことが困難であり，提案方式も対象としない．

以上で述べた，バグの種類と本論文で提案するデバッガとの関係をまとめると，次のようになる．

- リアルタイム部分，特にインタラクティブ性の高い部分における論理エラー（表 1 (a)）：提案方式がまさに有効である．
- リアルタイム部分のインタラクティブ性のあまり高くない部分における論理エラー

(表 1 (b))：プロファイラを利用するなどしても除くことができるが，提案方式も有効である．

- 非リアルタイム部分の論理エラー（表 1 (c)），および，実行時エラー（表 1 (d)）：既存デバッガを利用して除くことができ，提案方式は対象としていない．
- 再現性のないエラー：既存デバッガを利用しても除くのは困難であり，提案方式は対象としていない．

2.3　既存のデバッグ方法とその問題点

本節では，ゲームプログラム内のループ処理などのリアルタイム部分における論理エラーを，既存のデバッグ方法で発見しようとする際に生じる問題点について述べる．ここでは，既存のデバッグ方式として，デバッガを用いる方法，および，必要な情報を画面に出力してリアルタイムに観察する方法をとりあげる．

2.3.1　デバッガを用いる方法

Microsoft Visual Studio や Eclipse などの統合開発環境では，付属のデバッガを用いてデバッグを行うことが可能である．これにより，**図 4** のようにブレークポイントを設定し，プログラムを一時停止させてソースコード中の実行位置情報やプログラム内部の変数情報，メソッドの呼び出し履歴情報を表示することができる．

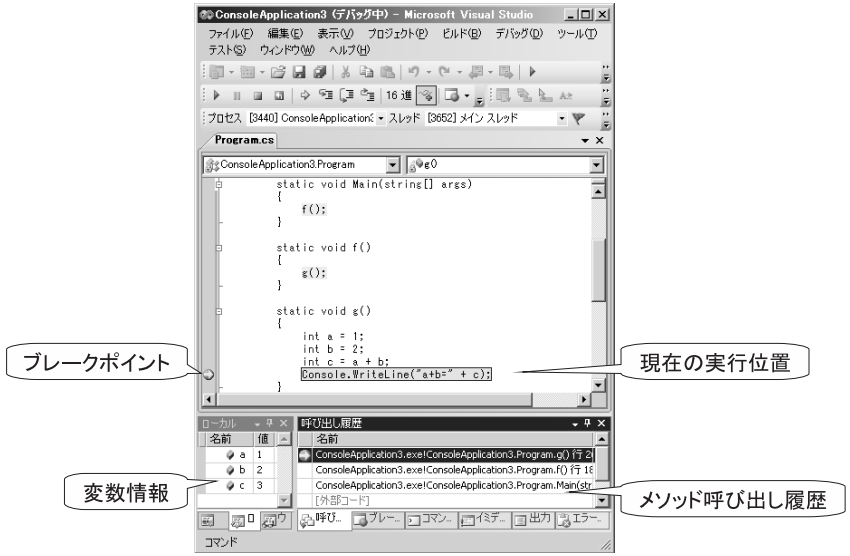

図 4　Microsoft Visual Studio 付属のデバッガ
Fig. 4　Microsoft Visual Studio debugger.

しかし，このような既存のデバッガでは，プログラム停止時点の情報を得ることはできても，どのような実行経路でそこに至ったかまでは分からない．プログラムの実行経路情報を得るために多数のブレークポイントを設置すると，プログラムが頻繁に停止するようになり，プレイヤのキー入力などが正常に行えなくなることもありうる．また，キャラクタの位置座標などの，フレームごとに刻々と変化していく変数情報をゲームプレイ中に確認しようとするたびにプログラムを停止させていては，デバッグの効率が悪くなってしまう．

ここで具体的な例として，プレイヤのキー入力によってキャラクタの移動を行ったり，アニメーションを設定したりするプログラムのデバッグを考えてみる．**図 5** (a) のように，`primeActor.setAnimation("walk")` にブレークポイントを設置したとする．この位置でプログラムが停止したとしても，`vx == 0.0 && vz == 0.0` が偽であったことしか分からず，`System.keyDown(System.VK_ W)` の真偽値はどうであったか，キー入力は正常に行われたのかということは分からない．しかも，このプログラムではキー入力をするたびに，`vx == 0.0 && vz == 0.0` が偽になり `primeActor.setAnimation("walk")` へ制御が移り，ブレークポイントにヒットしプログラムが中断され，デバッグの効率が悪くなる．

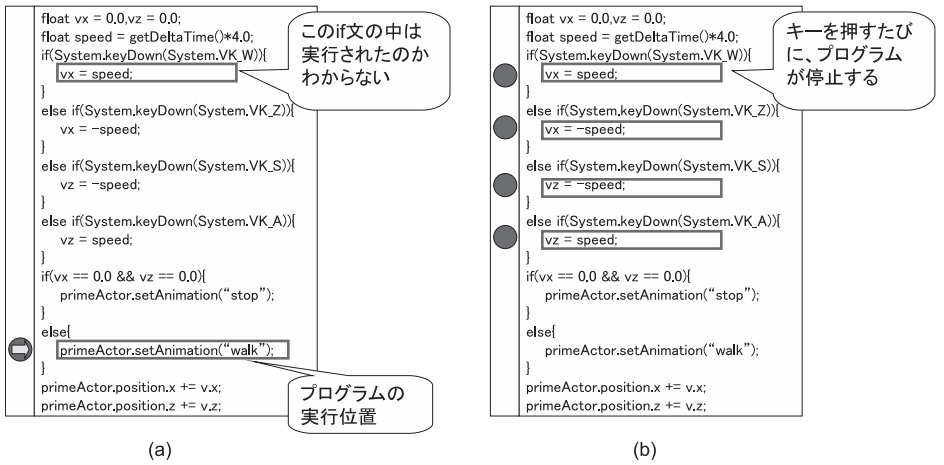

図 5　ブレークポイントによるプログラムの停止
Fig. 5　Program paused by breakpoints.

キー入力が正確に行われているか調べるため，図 5 (a) を変更し，(b) のように多数のブレークポイントを設置すれば，それぞれのキー入力が正しく行われているか確認することが可能であるが，キー入力が行われるたびにプログラムが中断されることに変わりはない．そのうえ，キー入力のタイミングや順番が重要である箇所では，プログラムが中断されてしまうことにより次のキー入力が正常に行えなくなり，ゲームプログラムを期待どおりに動作させることができなくなる可能性がある．たとえば，このプログラムで複数のキーを同時に押したときの挙動を調べるのは非常に困難である．なぜならば，同時に押したつもりでも，わずかなタイミングの差で最初に押したキーの判定が先に行われ，そこでブレークポイントにヒットし，プログラムが中断されてしまうからである．変数情報の取得に関しても，速度 `vx`，`vz` やキャラクタの位置座標 `primeActor.position` の値をデバッガで確認するためには，ブレークポイントなどを用いてプログラムを中断させる必要がある．

2.3.2　必要な情報を画面に出力する方法

リアルタイム部分の論理エラーを発見するためのもう 1 つの手段として，必要な情報を画面に出力する方法がある．この方法では，情報の更新がフレームごとに，つまりゲーム画面の再描画と同時に行われる．そのため，ゲームプレイを継続しながらプログラム内部の変数情報などを確認できるが，情報を表示するためのコードをソースコード中に埋め込む必要がある．

図 6 では，どのキーが押されたかの実行経路情報と，キャラクタの位置座標 `primeActor.position` の値をゲーム画面上に表示するコードをソースコード中に書き加えている．プレイヤがキー入力を行うと，ゲーム画面には押したキーの名前と，変化していく位置座標が表示される．このように，この方法ではゲームプレイを中断することなく，プログラム中の特定の箇所が実行されたかという情報や，1 フレームごとに変化していく変数の値をリアルタイムに観察することが可能ではあるが，プログラム中の随所にデバッグ情報表示用のコードを記述しなければならない．これは，本来デバッガでサポートされるべきことを手作業で行っているため，プログラムの可読性が下がるうえに，開発者にとって負担が大きい．

3.　ゲームプログラムに適したデバッガ

前章で述べたように，既存のデバッガではプログラムの実行位置情報やプログラム内部の変数情報を確認する際には，プログラムを一時停止させることが前提となっており，これがインタラクティブ性とリアルタイム性が高いゲームプログラム内のループ部分をデバッグするうえでの問題点となっている．そのため，ゲームプレイを続けながらプログラム内部の状態を観察する必要があるときには，観察したい情報をゲーム画面に表示するコードをプログラム中に手作業で記述するという負担を開発者は強いられる．

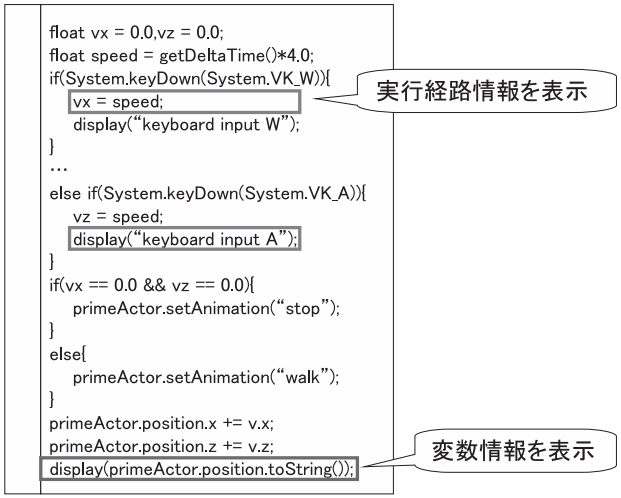

図 6　デバッグ情報表示用のコードを挿入
Fig. 6　Insert of debug code.

そこで本論文では，プログラムを停止させずにプログラムの実行経路情報，プログラム内部の変数情報を観察することができる，ゲームプログラムに適したデバッガを提案する．そのため，kameTL およびその統合開発環境上に，提案するデバッガを実現することを目指した．

以下，本章ではまず kameTL とその並行処理の記述方法について説明し，次にゲームプログラムに適したデバッガの設計とその機能の詳細について詳しく述べる．

3.1　kameTL

kameTL はゲームに特化した並行処理機構を持つ Java 風のオブジェクト指向言語である．kameTL では多数の同時並行に動くキャラクタの制御を容易に記述することができる．

3.1.1　ノンプリエンプティブ・スレッド

ゲームプログラムでは，1 フレームの間ですべてのキャラクタを少しずつ動かし画面に描画する処理を繰り返す．このことにより，ゲームをプレイしている者に，あたかも画面内で

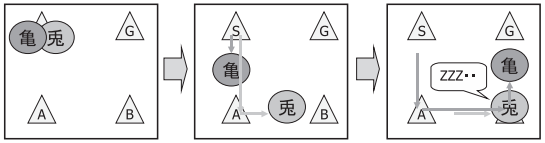

図 7　亀と兎の競争
Fig. 7　The race of turtle and rabbit.

```
//キャラクタの抽象クラス
class Actor extends CoThread{
    //位置座標
    private Point pos;
    //to は目的地の座標　 velocity は速度
    public void moveTo(Point to,int velocity){
        //目的地に達するまで繰り返す
        while(!pos.equals(to)){
            //位置座標 pos を 1 フレーム分更新
            updatePoint(pos,to,velocity);
            //1 フレーム分の処理が終わったら
            //スレッドを切り替える
            yield;
        }
    }
}
```

```
//亀
class Kame exnteds Actor{
    public void main(){
        moveTo(A,2);//A 地点へ
        moveTo(B,2);//B 地点へ
        moveTo(G,2);//G 地点へ
    }
}
//兎
class Usagi exnteds Actor{
    public void main(){
        moveTo(A,6);//A 地点へ
        moveTo(B,6);//B 地点へ
        sleep(60);//1 分間眠る
        moveTo(G,6);//G 地点へ
    }
}
```

図 8　亀と兎の競争の kameTL ソースコード
Fig. 8　kameTL source code of the race of turtle and rabbit.

たくさんのキャラクタが連続して動いているように見せることができる．よって，各キャラクタの一連の動作の処理でも，1 フレーム分の処理を終えたら途中で中断し，描画命令を呼ばなくてはならない．kameTL では，このような処理を実現するための機構としてノンプリエンプティブ・スレッド（以下，単にスレッドと呼ぶ）を備えている．kameTL のスレッドは 1 フレーム分の処理を終えると明示的に `yield` 文を用いて，次のスレッドに制御を渡す必要がある．

スレッドを用いてゲーム特有の並行処理を記述する例として，**図 7** のような亀と兎の競争を考える．亀は歩きながら，また兎は走りながら A 地点，B 地点を経由して G 地点まで移動する．ただし，兎は B 地点で 1 分間眠るとする．このような処理は kameTL では**図 8** のように書ける．1 フレーム分の処理を終えたスレッドの処理権限切替えには明示的に `yield` 文を用いる．そして，各スレッドは**図 9** のように，1 フレームの間に 1 回ずつ実行される．このように，kameTL では各キャラクタの一連の動作の流れを自然に記述することが可能である．

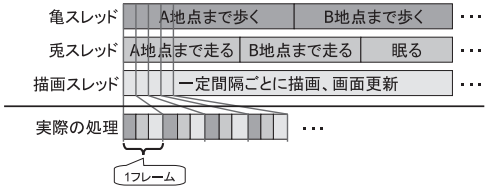

図 9　亀と兎の競争のスレッドの動作
Fig. 9　Behavior of threads of the race of turtle and rabbit.

3.1.2　スレッドの実行順序

1 フレームにおけるスレッドの実行順序は一意に定まる必要がある．たとえば，当たり判定用のスレッドは，すべてのキャラクタが 1 フレーム分移動し終わってから実行される必要がある．

kameTL のスレッドは，親子関係による木構造を形成する．この木構造をスレッド階層木という．kameTL はこのスレッド階層木に基づき，次のようにして 1 フレームにおけるスレッドの実行順序を一意に定めている．

(1)　スレッド階層木を深さ優先の順で実行する．
(2)　深さが等しい場合は，スレッドが持つ優先順位の値が大きい順で実行する．
(3)　深さと，スレッドが持つ優先順位の値がどちらも等しい場合は，生成された順で実行する．

スレッドの実行順序を決める際に，スレッドの優先順位の値だけではなく，スレッドの親子関係も利用しているのは，柔軟にスレッドの実行順序を決められるようにするためである．

3.2　設　　計

プログラムを停止させず，1 フレームごとにプログラムの実行経路情報とプログラム内部の変数情報を表示することを，本デバッガを設計するうえでの基本方針とした．つまり，ゲーム画面の再描画と同期して，これらのデバッグ情報も更新されるということである．

図 3 のように，ゲームプログラムでは特定の処理がフレームごとに繰り返し行われている．そのため，プログラムの実行経路情報やキャラクタの位置座標などのプログラム内部の

変数情報は，1 フレームを境に内容が変わることが多い．すなわち，ゲームプログラムでは 1 フレームという処理の切れ目でこれらの情報を更新し，表示するようにすれば，開発者にとっては有益な情報が得られると考えられる．

プログラムの実行経路情報は，1 フレームの間にプログラムが実行したソースコード中の行を，時間的に前に実行された行ほど薄く，後に実行された行ほど濃くグラデーション表示し，これをフレームごとに更新する．このように，1 フレームの間にプログラムが実行していった跡をグラデーション表示させることにより，多数のブレークポイントを設置することなく，実行経路情報を視覚的に把握しやすくしている．本デバッガの実装の詳細は，4 章で詳しく述べる．

3.3　画面構成と機能

本デバッガの画面構成を**図 10** に示す．以下に実際にゲームプレイを継続しながら，リアルタイムにキー入力部分のデバッグを行う例を用いて，本デバッガの各機能について述べる．

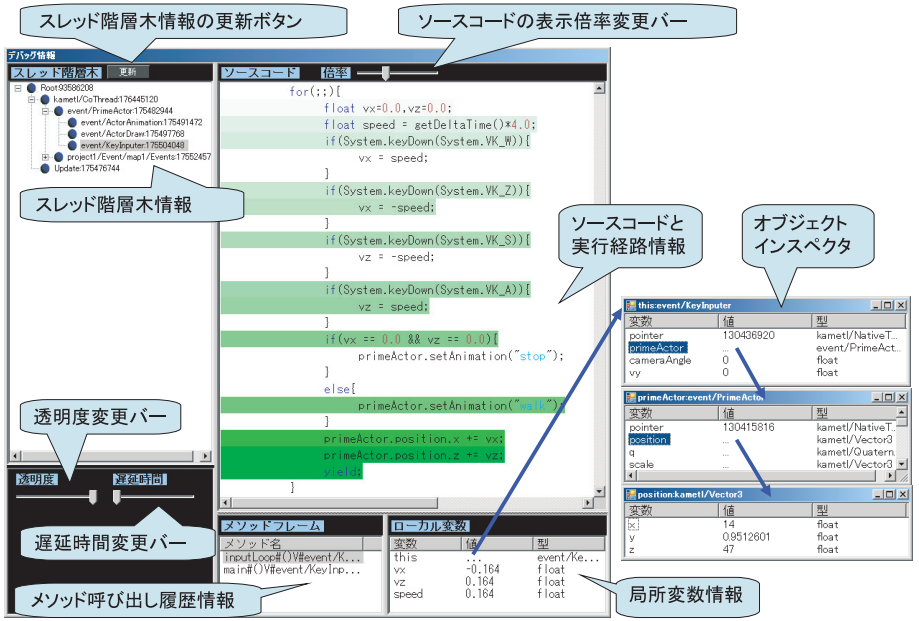

図 10　デバッガの画面構成
Fig. 10　Debug window.

3.3.1　スレッド階層木情報

画面左上には，スレッド階層木の情報がツリービューで表示される．ツリービューの各ノードがスレッドを表す．スレッドインスタンスのクラス名と生成時に割り振られた ID の組がノード名になっており，観察したいスレッドを見つけやすいようになっている．この中から，観察したいスレッドをクリックすると，そのスレッドのリアルタイムな情報更新が始まり，そのスレッドのメソッド呼び出し履歴情報が，リアルタイムに更新され表示されるようになる．なお，本デバッガでは特に重要な情報であるプログラムの実行経路情報とプログラム内部の変数情報をリアルタイムに観察することを基本方針としているため，スレッド階層木のリアルタイムな情報更新は行わない．スレッド階層木の情報を更新したい場合には，スレッド階層木情報の上にある更新ボタンを押して明示的に情報を更新する必要がある．

この例ではプレイヤのキー入力を行っている `event/KeyInputer` クラス（以下，単に `KeyInputer` クラスと呼ぶ）のインスタンスであるスレッドを選択している．

3.3.2　メソッド呼び出し履歴情報

画面下中央には選択したスレッドのメソッド呼び出し履歴情報が表示され，リアルタイムに更新される．ここに表示されるメソッドの中から観察したいメソッドをクリックすると，そのメソッドのソースコードとその実行経路情報，そしてローカル変数情報も表示，更新されるようになる．この例では，選択した `KeyInputer` クラスのインスタンス（ID は 175504048）であるスレッドのメソッド呼び出し履歴情報が表示されており，キー入力が行われている `KeyInputer` クラスの `inputLoop` メソッドを選択している．

3.3.3　ソースコードと実行経路情報

画面右側には，選択したスレッドが 1 フレームの間に実行してしたソースコード中の行が，時間的に前に実行された行は薄い緑色，後に実行された行は濃い緑色というようにグラデーション表示され，フレームごとに更新される．また，1 度実行された行の色はすぐには消えず，約 1 秒間かけてフェードアウトしながら消えていく．これは一瞬しか通らなかった行を見落とすことがないようにするためである．

図 10 は，キー入力を扱う `inputLoop` メソッドのソースコード上で `KeyInputer` クラスのインスタンスであるスレッドが 1 フレーム間に実行していった跡を表示している．この実行跡を一目見るだけで，プレイヤは Z キーと A キーを押しており，`System.keyDown(System.VK_Z)` と `System.keyDown(System.VK_A)` が真となり，それらの if 文の中が実行されているということが分かる．

これら実行経路情報の表示は，ゲームプレイを中断することなく，リアルタイムに行われ

る．そのため，開発者はキー入力などインタラクティブな操作によって，プログラム内のどの部分が実行されたのかを多数のブレークポイントを設置せず容易に把握することが可能になる．このようなキー入力のほかにも，当たり判定や敵キャラクタに関するルーチンなどの処理に対してもリアルタイムな実行経路情報の表示はきわめて有効である．

3.3.4　ソースコードの表示倍率変更バー

ソースコードの上部にある表示倍率変更バーを用いることで，ソースコードの表示倍率を変更することができる．観察対象のメソッドの定義が長く，プログラムの実行経路も長かった場合，ソースコード表示用のウィンドウ内に，すべての実行経路情報が収まりきらなくなることがある．実行経路情報の全体をとりあえずおおまかに把握したい場合には，ソースコード表示倍率を小さくし，ウィンドウ内にすべての実行経路情報を収めることができる．

3.3.5　局所変数情報

画面右下には，選択されたメソッドの局所変数情報が表示され，リアルタイムに更新される．`int` 型などのプリミティブ型の変数，そして `String` 型など一部の参照型に関しては，その値が局所変数情報表示欄に，変数名や型と一緒に表示される．一般的な参照型に関しては，クリックすることによりそのインスタンス変数のオブジェクトインスペクタを開き，メンバ変数を観察することもできる．ここでは，`inputLoop` メソッドの局所変数である `float` 型の `vx`，`vz` などが表示されており，これらの値がプレイヤのキー入力によって変化していく様子がゲームプレイを続けながら観察できる．

1つ留意しておくべき点は，観察している最中に選択しているメソッドから呼び出し元に戻ってしまい，表示されている局所変数が無効になってしまう可能性があるということである．この例では，仮にプログラムの制御が `inputLoop` メソッドから呼び出し元の `main` に戻ったとすると，観察していた `inputLoop` の局所変数はすべて無効になる．そのため，プログラムの制御が観察対象のメソッドがらはずれた際には，局所変数情報の表示欄は非アクティブ状態になり．観察対象のメソッドに再び制御が移ると，再びアクティブ状態に戻るようにした．

3.3.6　オブジェクトインスペクタ

選択されたインスタンス変数のメンバ変数が表示され，リアルタイムに更新される．参照型のメンバ変数に関しては，さらにその変数のオブジェクトインスペクタを再帰的に開いていくことも可能である．この例では，`inputLoop` メソッドの局所変数である `this` のオブジェクトインスペクタを開き，そこからさらに観察したい変数のオブジェクトインスペクタを開いていくことにより，最終的に三次元ベクトルである `primeActor.position` のメンバ変数 `x`，`y`，`z` の値が，プレイヤのキー入力によってリアルタイムに変化していく様子を観察している．

3.3.7　透明度変更バー

画面左下にある透明度変更バーを用いれば，デバッグウィンドウの透明度を変更し，ゲーム画面と重ねて両方の画面を同時に見ることができる．本デバッガは，ゲームプレイを継続しながら同時にデバッグを行うことを前提としているため，ゲーム開発者はゲーム画面とデバッグ画面を同時に見なければならない．しかし，単一ディスプレイの環境では2つのウィンドウを同時に効率良く見るのは困難であり，特にフルスクリーンでのデバッグのときは不可能に近い．そこで，デバッグウィンドウを半透明にし，ゲーム画面に重ねて表示することによってこの問題を解決した．

キーイベントは，どのウィンドウにフォーカスがあってもゲーム画面で発生するようにした．そのため，半透明のデバッグウィンドウが完全にゲーム画面を覆っていても，キー入力は必ずゲーム画面で受け付ける．また，マウスイベントはフォーカスのあるウィンドウで発生するようにした．したがって，マウスイベントをゲーム画面で受け付けながらデバッグを行いたい場合には，デバッグウィンドウがゲーム画面に完全に重ならないようにし，ゲーム画面にフォーカスがいくようにする必要がある．

3.3.8　遅延時間変更バー

透明度変更バーの右側にある．フレームごとに遅延を挟むことができ，これを用いればゲームのスピードを遅くすることができる．シューティングゲームやアクションゲームなど，比較的キャラクタの動きが速いゲームでは，通常の速さでゲームプレイしていると，処理が速すぎて，デバッグ情報を追い切れない場合もある．そこで，フレームの間に適度な遅延を入れることにより，意図的にゲームのスピードを遅くし，デバッグ情報を追いやすくすることができるようにした．バーを一番右に移動させると，ゲームを完全に停止させることもできる．

4. 実　　装

提案するデバッガを kameTL および，その統合開発環境上に実装した．本デバッガは**図 11** に示すように，kameTL 仮想機械，デバッグ API，そしてデバッグウィンドウで構成されている．kameTL 仮想機械部分は C++で作成し，デバッグウィンドウ部分は C# で作成した．

本デバッガは，フレームごとに kameTL 仮想機械からデバッグ API を通じて取得した内

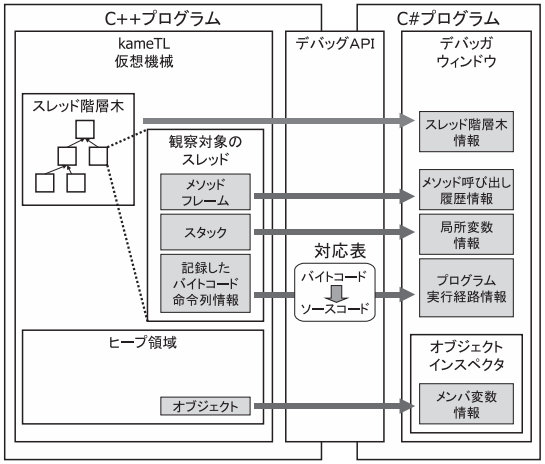

図 11　デバッガの構成
Fig. 11　Composition of debugger.

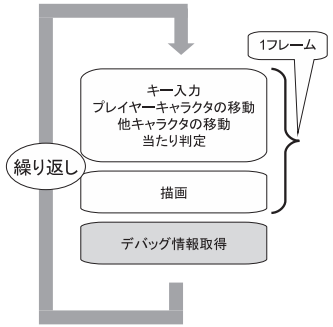

図 12　デバッグ情報更新
Fig. 12　Update of debug information.

部情報を，デバッグウィンドウに表示する．プログラムの実行経路情報の取得については，観察対象のスレッドのみ，フレームごとに実行したバイトコード命令列を記録することで実現している．ゲームの進行とデバッグ情報の更新がフレームごとに行われるイメージは**図 12** のようになる．以下に，kameTL 仮想機械のデバッグ API によるデバッグ情報取得方法の詳細について述べる．

4.1　kameTL 仮想機械

kameTL のプログラムは，kameTL コンパイラにより kameTL 仮想機械のバイトコードへコンパイルされる．kameTL 仮想機械は，Java 仮想機械[2),3)] に類似したスタックマシンになっており，バイトコードを読み込んで動作する．バイトコードの仕様もほぼ Java と同じである．ただし，kameTL 仮想機械では Just In Time（JIT）方式は用いず，バイトコードをすべて解釈実行する．

4.1.1　スレッド階層木

kameTL 仮想機械は内部にスレッド階層木を持っている．そして，1 フレームごとにこれらのスレッドを深さ優先の順で 1 度ずつ実行していく．基本的には最後に実行されるスレッドがゲーム画面への描画処理を行う．

4.1.2　スレッド

kameTL のスレッドはプログラムのコンテキストを管理するためのプログラムカウンタ，スタック，メソッドフレームを持っている．スレッドは自分に実行処理権限が回ってくるとプログラム中の前回停止した位置から，プログラムの実行を再開する．そして，`yield` 文（バイトコードレベルでは `yield` 命令）のある位置で再び停止し，次のスレッドに処理権限を移す．これが各スレッドの 1 フレームの処理にあたる．

4.1.3　バイトコード命令列の記録

プログラムの実行経路情報を表示するためには，フレームごとに実行されたバイトコード命令列を記録する必要がある．そのため，kameTL 仮想機械のバイトコード命令実行処理部分で，各バイトコード命令を実行した後に，そのときのプログラムカウンタとメソッド名の組を毎回記録するようにした．このようなバイトコード命令列の記録は観察対象のスレッドに関してだけ行われるため，大きなオーバヘッドとはならない．

4.2　デバッグ API

kameTL 仮想機械内部の状態を取得するためのインタフェースである．デバッグ API は，C++/CLI [⋆1]で記述した．これは，C++で記述された kameTL 仮想機械の情報を C# で記述されたデバッグウィンドウに渡せるようにするためである．

kameTL の主要なデバッグ API を**表 2** に示す．デバッグ API による kameTL 仮想機械内部の情報取得は，必ずフレームとフレームの間に行われるため，このときすべてのスレッ

⋆1 .NET Framework の共通言語基盤（CLI）上で実行するプログラムを作るための，C++を拡張したプログラミング言語．

表 2　デバッグ API
Table 2　Debug API.

VirtualMachine クラス	
CoThreadInfo GetRootCoThreadInfo()	スレッド階層木のルートスレッドの情報を取得
CoThreadInfo クラス	
List<CoThreadInfo> GetChildren()	子スレッドのリストを取得
void SetExecuteMode(int kind)	1 にするとバイトコード命令列を回収するようになる（通常は 0）
List<ExecutedCode> GetExecutedCodes(string fileName)	記録したバイトコード命令列を取得
List<MethodFrameInfo> GetMethodFrames()	メソッドフレームのリストを取得
string GetThreadId()	スレッド ID を取得
string GetThreadClassName()	スレッドインスタンスのクラス名を取得
ExecutedCode クラス	
string GetFileName()	命令列に対応するソースコードの名前を取得
int GetLine()	命令列に対応するソースコード中の行番号を取得
MethodFrameInfo クラス	
List<VariableInfo> GetLocalVariables()	局所変数のリストを取得
string GetMethodName()	メソッド名を取得
string GetClassName()	メソッドが定義されているクラス名を取得
VariableInfo クラス	
VariableInfo[] GetMembers(int index)	指定した番号の局所変数のメンバ変数リストを取得
bool HasMembers()	参照型であり，メンバ変数を持っていれば真
VariableInfo[] GetMembersList()	メンバ変数を取得
string GetName()	変数名を取得
string GetTypeName()	型名を取得
string ToString()	値の文字列表現を取得

ドは `yield` 命令で停止した状態になっており，安全にデバッグ情報を取得できる．以下，デバッグ API を用いて各種デバッグ情報を取得する方法を説明する．

4.2.1　スレッド階層木情報の取得

kameTL 仮想機械へのインタフェースである `VirtualMachine` クラスの `GetRootCoThreadInfo` メソッドで，スレッド階層木のルートスレッドの情報を取得できる．スレッド情報は `CoThreadInfo` クラスで表現されている．そして，このクラスの `GetChildren` メソッドで子スレッド情報をすべて取得できるため，子スレッド情報の取得を再帰的に繰り返すことで，すべてのスレッドとその親子関係の情報を取得できる．

4.2.2　スレッド情報の取得

`CoThreadInfo` クラスの `GetThreadId` メソッドでスレッドの ID を取得でき，`GetThreadClassName()` メソッドでスレッドインスタンスのクラス名を取得できる．また，スレッドの実行モードは `SetExecuteMode` メソッドで切り替えることが可能である．引数を 1 にするとデバッグモードになり，スレッドは 1 フレームの間に実行したバイトコード命令列を記録するようになる．そして，引数を 0 にすると通常モードになり，バイトコード命令列の記録は行わなくなる．このスレッドのモード切替え機能を使うことにより，観察対象のスレッドに関してだけバイトコード命令列を記録するようにでき，オーバヘッドを抑えられる．

4.2.3　実行経路情報の取得

`CoThreadInfo` クラスの `GetExecutedCodes` メソッドで，デバッグモードで動いたスレッドが 1 フレーム間に実行したバイトコード命令列を取得できる．バイトコード命令を表現する `ExecuteCode` クラスには `GetFileName` メソッドと `GetLine` メソッドがあり，命令コードに対応するソースファイル名と行番号を取得できる．命令コードはプログラムカウンタとメソッド名の組である．この組からソースコードのファイル名と行番号の組へと変換する対応表は，バイトコードにデバッグ情報として付加されている．

4.2.4　メソッド呼び出し履歴情報の取得

`CoThreadInfo` クラスの `GetMethodFrames` メソッドで，スレッドが保持しているメソッドフレーム情報 `MethodFrameInfo` クラスのリストを取得できる．さらに，`MethodFrameInfo` クラスの `GetMethodName` メソッド，`GetClassName` メソッドで，それぞれのメソッドの名前と定義されているクラス名が分かる．そして，これらの情報からメソッド呼び出し履歴情報が作成できる．

4.2.5　局所変数情報の取得

`MethodFrameInfo` クラスの `GetLocalVariables` メソッドで，メソッドフレームが表すメソッドのスコープ内の局所変数情報を取得できる．変数情報は `VariableInfo` クラスで表現されており，`GetName` メソッド，`GetTypeName` メソッド，`ToString` メソッドでそれぞれ，変数名，型名，値の文字列表現を取得できる．

4.2.6　メンバ変数情報の取得

`VariableInfo` クラスの `HasMembers` メソッドで，その変数がメンバ変数を持っているかを調べることができる．メンバ変数を持つ場合は `GetMembers` メソッドでメンバ変数のリストを取得する．この機能を使うことにより，局所変数情報からオブジェクトインスペクタを開き，そこからさらに新しいオブジェクトインスペクタを再帰的に開くといったことが実現できる．

5. 活 用 事 例

ここでは，kameTL で記述された 2 つのゲームプログラムを例にとり，具体的活用事例を通して本デバッガの有効性を示す．1 つ目のロールプレイングゲームの例では，主人公が城にいる兵士と会話をするプログラムのデバッグについて述べる．これは表 1 (a) に相当するバグの例である．2 つ目のシューティングゲームの例では，敵が発射する弾が描く弾道の種類を，乱数で決めるプログラムのデバッグについて述べる．これは，表 1 (b) に相当するバグの例である．

5.1 ロールプレイングゲーム

ここでのロールプレイングゲームでは，主人公キャラクタを操作して町や城にいる他のキャラクタたちと会話して情報を集め，モンスターを倒してレベルを上げるということを繰り返してゲームを進める．主人公キャラクタが他のキャラクタに話しかける部分は，プレイヤの操作に対して他のキャラクタが反応して行動を起こすという，インタラクティブな処理である．

たとえば，主人公が近くにきて話しかけると「私に話しかけるな！」と返事をし，ジャンプをすると「私のそばで跳ねるな！」という兵士の処理を記述し，そのデバッグを行うことを考えてみる．プログラムの擬似コードは**図 13** のようになるだろう．

この処理を実際にプログラムとして記述し，動かしてみたところ話しかけたときの処理はうまくいったが，**図 14** のように，兵士の近くで主人公がジャンプしたときに，兵士が何も反応しないというバグが発生した．このときのプログラム実行経路情報を**図 15** に，プログラム内部の変数情報を**図 16** にそれぞれ示す．

```
for(;;){
    if(主人公と兵士の座標が近い){
        if(主人公が話しかけるボタンを押した){
            「私に話かけるな！」と表示する
        }
        if(主人公の y 座標が,
           兵士の y 座標より大きい){
            「私のそばで跳ねるな！」と表示する
        }
    }
}
```

図 13　兵士の動作の擬似コード
Fig. 13　Pseudo code of soldier's action.

まず，実行経路情報を見ると `d <= 1.5`（ここで `d` は主人公と兵士間の距離）の部分は真になり，if 文の中へと制御が移っている様子が分かる．これは局所変数情報において `d` の値が 1.055503 となっており，1.5 以下であることからも分かる．これでバグの原因はこの if 文の中にあることが分かった．`if(primePos.y >= position.z + 0.3)` の部分をよく見てみると，兵士の y 座標を取得すべきところで `position.z` と誤って z 座標を取得してしまっており，これがバグの原因であることが分かる．さらにオブジェクトインスペクタの値を見ると，主人公の y 座標である `primePos.y` は 1.95492，兵士の y 座標である `position.y` は 1.257025 となっているため，その差は 0.3 より大きい．つまり，`position.z` を `position.y`

図 14　ロールプレイングゲームの一画面
Fig. 14　A screen of a roll playing game.

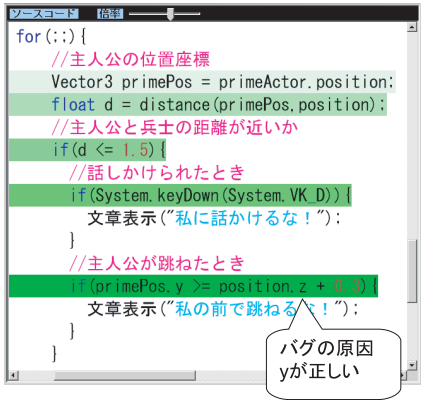

図 15　当たり判定処理部分の実行経路情報
Fig. 15　Flow of execution in collision part.

図 16　プログラム内部の変数情報
Fig. 16　Values of variables.

と書き直せば，おそらく意図したとおりに動作し，「私のそばで跳ねるな！」と表示されるであろうことも推察できる．

注目すべき点は，このようなデバッグ作業がすべてゲームプレイを継続しながら行えているということである．プレイヤがキー入力を行い，主人公を操作すると，それにともない主人公の位置座標の値や，兵士との距離 d の値が変わっていく様子がリアルタイムに観察できる．そして，主人公が兵士に近づいたり，話しかけたり，そばで跳ねたりするたびに，兵士を制御するスレッドの実行経路情報も刻々と変化し，その様子がリアルタイムにデバッグ情報として表示される．

ロールプレイングゲームにおける，これ以外のインタラクティブな処理の例としては，モンスターとの戦闘場面におけるキー操作によるコマンド入力部分などがあげられる．その他にも，アニメーション制御部分，登場人物であるキャラクタの自律的な移動制御部分などでは，ゲームプログラムを動かし続けながらデバッグを行うことが有効であろう．

```
//1 秒おきに 20 発の弾を撃つ
public void shootAll(){
    for(;;){
        for(int i=0;i<20;++i){
            shoot();
        }
        sleep(1.0);
    }
}
```

```
//10 種類の軌道から，ひとつ選んで撃つ
public void shoot(){
    int rnd = Math.random() % 10000;
    if(rnd <= 1200){
        shoot1();
    }
    else if(rnd <= 2400){
        shoot2();
    }
    ...
    else if(rnd <= 8600){
        shoot9();
    }
    else if(rnd > 8600){
        shoot10();
    }
}
```

図 17　シューティングゲームのソースコードの一部
Fig. 17　Program fragment of a shooting game.

5.2　シューティングゲーム

ここでのシューティングゲームでは，プレイヤが敵からの攻撃をかわしながら，相手に向かって弾を発射し当てて倒すことでゲームを進めていく．敵や，敵が発射する弾の動きには一定のパターンがあり，このパターンを工夫することによりゲームの難易度が上がったり，面白くなったりする．

たとえば，1 秒ごとに 20 発の弾をまとめて撃ってくる敵を作るとする．敵の弾の軌道は 10 種類あり，この 20 発の弾の軌道はそれぞれこの 10 種類の軌道から乱数で選んで決めることにする．つまり，敵は 1 秒おきに様々な軌道を持つ 20 発の弾を撃つわけである．このプログラムのコードは**図 17** のようになる．`shoot` メソッドでは `Math.random` メソッドで乱数を取得し，それを利用して 10 種類の軌道のうちの 1 つを選び発射するようにした．

このプログラムを実際に動かしてみたところ，確かに 1 秒おきに様々な弾道の弾を敵が発射したので，プログラムは正しく動いているように思えた．しかし，本デバッガで `shoot` の実行経路情報をしばらく観察したところ，**図 18** の (a)，(b) のように `shoot5` メソッドがまったく実行されていないことが分かった．よく見てみると，**図 19** の `if(rnd <= 4200)` の部分が誤りであることが分かる．`if(rnd <= 4200)` の上の条件文は `if(rnd <= 4300)` となっているため，`if(rnd <= 4200)` が実行されることはないのである．ここは，4300 と

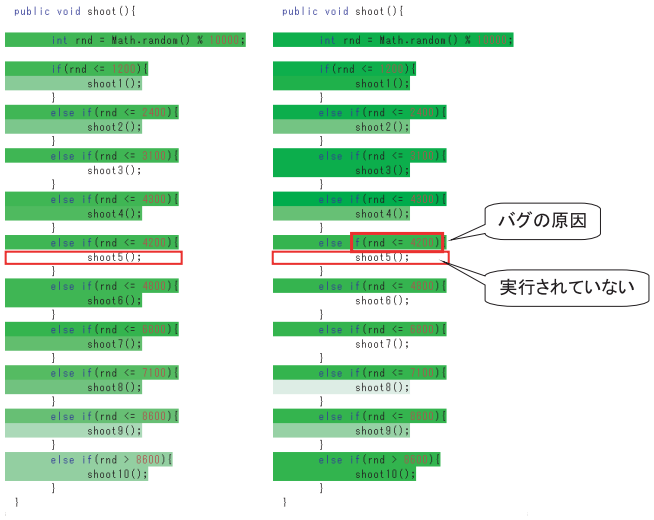

図 18　シューティングゲームの実行経路情報
Fig. 18　Flow of execution of a shooting game.

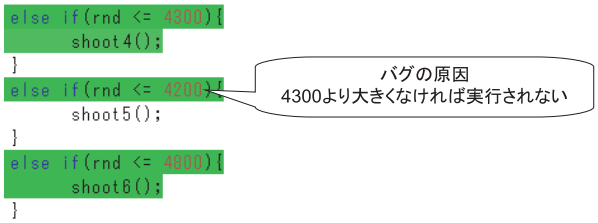

図 19　シューティングゲームのプログラムのバグ
Fig. 19　Bug in the program of a shooting game.

4800 の間の数にするのが正しい．

20 発の乱数によって弾道の決まる弾が，実際に正しく動作しているかを見ただけで判断するのはきわめて困難である．本デバッガの，リアルタイムな実行経路情報の表示があったからこそ簡単に発見できたバグである．このように，通常のデバッガでは停止したプログラムの状態しかデバッグ情報として用いることができなかったが，本デバッガでは刻々と移り変わっていくプログラムの状態をデバッグ情報として使えるのである．

このバグの検出にはプロファイラを用いて，ソースコード中の各行の実行回数をカウントしておくという方法もある．しかし，実行経路情報がリアルタイムにソースコード上に反映され，即座に確認できるという点において，本デバッガを用いた方が視認性が良くデバッグ効率が良い．

今回紹介したバグはいずれも単純ではあるものの．一見しただけでは気付かないことが多い．乱数を使用しているため，プログラムの動作が不規則であることがバグを見つけにくくしている原因である．ゲームプログラムでは乱数を多用する．たとえば，アクションゲームにおける敵キャラクタの動きや，ロールプレイングゲームにおけるモンスターの行動パターンルーチン，ボードゲームならばサイコロの出る目など乱数はいたるところで使われる．そのような乱数が使用されている箇所で，ゲームを動かし続けながらプログラムの実行経路の推移を観察することは，表面に出てきにくいバグを未然に防ぐことに役立つ．

6. オーバヘッド

提案したデバッガでは，フレームごとに観察対象のスレッドのバイトコード列を記録し，それをソースコード上にグラデーション表示させたり，局所変数情報を取得して更新したりするといった処理を行う．そのオーバヘッドを測定する実験を行った．

計測用のプログラムとしては，5 章で紹介したシューティングゲームを少し変更したものを用いる．出現する敵の数を 800 体から 2000 体まで変えながら，通常のゲームプレイを行う場合と，ゲームプレイをしながら，キー入力用のスレッドを半透明で全画面表示したデバッグウィンドウで観察する場合について，1 フレームあたりの平均処理速度の計測を行った．環境は CPU が Intel Core2 Duo 3.0 GHz，メモリは 2 GB，ビデオカードは GeForce8600GT，そして OS は WindowsXP Professional SP2 である．結果を**表 3** に示す．

この結果を見ると，デバッガのある場合とない場合の 1 フレームあたりの処理時間は 8 ms 前後とほぼ一定であり，敵キャラクタの数を増やすほどデバッグ情報の更新と表示によるオーバヘッドは相対的に小さくなっていることが分かる．つまり，スレッドが増えて描画処理などの割合が増えると相対的にデバッガによるオーバヘッドは減るということである．本デバッガでは観察対象のスレッドに関してのみ実行経路情報の取得や局所変数情報の更新を行うため，プログラム全体でスレッドの数が増えてもそのせいでオーバヘッドが増えることはない．

表 3　デバッガによるオーバヘッド
Table 3　Overhead of proposed debugger.

敵キャラクタの数	800	1200	1600	2000
デバッガなし（ms）	17.08	25.06	32.18	39.52
デバッガあり（ms）	25.09	33.55	40.04	47.06
オーバヘッド（%）	46.9	33.9	24.4	19.0

7. 関 連 研 究

The Sun Studio に付属する Performance Tools[4] や CLOVER[5] 等のプロファイラは，ソースコードの各関数や各行が使用した CPU 時間等を計測してソースコード上に重畳表示するツールである．結果は，パフォーマンスの改善や，到達不可能箇所のバグ検出等に利用することができる．しかし，本デバッガのように，プログラムの実行を続けながら，キー入力等のインタラクティブな操作に対応し，プログラムの実行経路情報や変数情報がリアルタイムに変化する様子を監視することはできない．

Seesoft[6] は，プログラムに関する全体情報の把握を目的としたシステムで，プログラムの各行は色の付いた 1 本の線として表示される．この色はコードの変更履歴といった統計データを表し，ユーザはプログラムの概略を把握するとともに，プログラムの履歴情報を獲得することができる．本デバッガのように，リアルタイムにプログラムの実行経路情報を表示する機能はないが，全体情報の把握に使われている手法は本デバッガにも応用可能と考えられる．

ETV[7] は，プログラムを実行させ，その実行経路情報をファイルに記録し，プログラムを実行させた後にその情報を視覚的に表示するツールである．このツールは，プログラムの実行経路情報をデバッグに利用している点では本デバッガと似ている．しかし，本研究はインタラクティブ性，リアルタイム性の高いゲームプログラムを対象としており，プログラムの実行中に，1 フレームごとのプログラムの実行経路情報を更新，表示するという点で ETV とは異なる．ETV をゲームプログラムに適用しても，インタラクティブな操作により，プログラムの実行経路情報がどのように変化したかを即座に確認することはできない．

Haskell Program Coverage[8] は，Haskell のためのデバッグ機構であり，ソースコード中でプログラムが 1 度も評価しなかった箇所を黄色，評価されたときに毎回真だった箇所を緑，評価されたときに毎回偽だった箇所を赤，というように色分けして表示する．評価（実行）された箇所を色分けして表示する点で本デバッガと似ているが，プログラム実行中にリアルタイムにソースコードの実行中の位置を表示するものではない．

GUI を持つプログラムの理解支援のための可視化システム[9] は，Java における GUI を持つプログラムの動作理解を支援するシステムである．このシステムでは，マウスのクリックやキー入力に対応した，ソースコード中の実行位置を強調表示するという機能がある．しかし，このシステムはゲームプログラムを対象として作られたものではなく，1 フレームの間に実行した跡をリアルタイムにグラデーションさせながら強調表示したり，変数の値を監視したりといったことはできない．

西森らのアクションゲーム記述に特化した言語[10]，また，長の Tonyu[11] は kameTL と同様にゲームシステム記述のための並行処理機構として，ノンプリエンプティブ・スレッドを備えたプログラミング言語である．西森らの言語は，デバッグ機構を備えていない．Tonyu は，デバッグ機構としてプログラム内部の変数情報をリアルタイムに観察する機能は持つが，プログラムの実行位置情報をリアルタイムに表示する機能はない．

8. 結　　論

本論文では，ゲームプログラムを停止させずにプログラムの実行経路情報とプログラム内部の変数情報を観察することができる，リアルタイム性の高いデバッガを提案し，kameTL およびその統合開発環境上に実装を行った．本デバッガを用いることにより，開発者は多数のブレークポイントを設置し，何度もゲームプレイを中断することなく多くの情報が得られるようになる．本デバッガで用いたプログラムの実行経路情報をグラデーション表示させる手法は C# や Java，Ruby など kameTL 以外の言語にも応用可能と考えられる．

さらに，kameTL で記述されたロールプレイングゲーム，シューティングゲームを例にとり，当たり判定や乱数による条件分岐における本デバッガの活用事例および，その有効性を示した．

アクションゲームなど他のジャンルのゲームにも本デバッガの適用範囲を広げていくことが今後の課題といえる．

謝辞　本論文を執筆するにあたり，終始手厚いご指導をしてくださった電気通信大学の岩崎英哉教授，大山恵弘准教授，島根大学の鈴木貢准教授に感謝いたします．また，本デバッガは未踏ソフトウェア創造事業未踏ユースのプロジェクト “みんなで創る RPG” における RPG 共同創作環境の一部として開発しました．ご支援いただいております独立行政法人情報処理推進機構（IPA）の諸氏，東京大学の竹内郁雄教授，株式会社オープンテクノロジーズの皆様，共同開発者の唐澤雄気君，川ノ上哲規君にお礼申し上げます．

参 考 文 献

1) 丹野治門：ゲームシステム記述言語 kameTL におけるリアルタイムデバッグ機構，第 49 回プログラミング・シンポジウム報告集，pp.17–24 (2008).
2) Meyer, J. and Downing, T.: *Java Virtual Machine*, Java Series, O'Reilly and Associates (1997).
3) Tim Lindholm, F.Y.: *The Java Virtual Machine Specification*, Java Series, Addison-Wesley Pub. (1997).
4) Performance Tools. http://developers.sun.com/solaris/articles/perftools.html
5) CLOVER. http://www.atlassian.com/software/clover/
6) Elick, S.G., J.L.S. and Sumner, J.E.E.: Seesoft-A Tool For Visualizing Line Oriented Software Statistics, *IEEE Trans. Softw. Eng.*, Vol.18, No.11, pp.957–968 (1992).
7) Terada, M.: ETV — A Program Trace Player for Students, *Proc. 10th annual SIGCSE conference on Innovation and technology in computer science education* (*ITiCSE2005*), pp.118–122 (2005).
8) Gill, A. and Runciman, C.: Haskell Program Coverage, *Proc. ACM Haskell Workshop* (*Haskell2007*), pp.1–12 (2007).
9) 佐藤竜也，志築文太郎，田中二郎：GUI を持つプログラムの理解支援のための可視化システム，日本ソフトウェア科学会第 24 回全国大会講演論文集 (2007).
10) 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，Vol.44, No.SIG 15(PRO19), pp.36–54 (2003).
11) 長　慎也：Tonyu—アニメーション作成に特化したプログラミング言語と開発ツール，夏のプログラミング・シンポジウム，pp.99–103 (2001).

(平成 20 年 2 月 17 日受付)
(平成 20 年 4 月 28 日採録)

丹野　治門（学生会員）

2007 年電気通信大学電気通信学部情報工学科卒業．2008 年現在，電気通信大学大学院電気通信学研究科情報工学専攻博士前期課程在学．2007 年度未踏ソフトウェア創造事業未踏ユースにおいて，情報処理推進機構よりスーパークリエータと認定される．プログラミング言語，デバッガ等に興味を持つ．