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

丹野 治門
tanno@ipl.cs.uec.ac.jp
電気通信大学 大学 院情報工学専攻

Real-time Debugger in Game Oriented Programming Language kameTL

Haruto Tanno
Department of Computer Science, The University of Electro-Communications

概要

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，プログラムを一時停止させてしまうようなデバッグ方法は向いていない．例えば，プレイヤー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てしまうと期待通りの動きをしなくなってしまう．また，ゲーム内のキャラクタの座標を見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．

そこで，本研究ではプログラムを停止させず，リアルタイムにデバッグを行える機構を，ゲームシステム記述言語 kameTL 及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もつ Java 風のオブジェクト指向言語である．本システムを用いることにより，開発者はゲームを中断することなく，ゲームプレイを続けた状態でデバッグを行うことができる．例えばプレイヤーがキー入力をし，その入力によってゲーム内のキャラクタの処理が分岐して実行されていく様子が，ゲームプレイを中断することなく確認できる．また，キャラクタの座標などがリアルタイムに更新されていく様子を観察することも可能である．

1 はじめに

ゲームプログラムのデバッグは難しい．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方法は向いていない．多数のキャラクタ同士の相互作用や，プレイヤー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てしまうと期待通りの動きをしなくなる可能性がある．また，ゲーム内のキャラクタの位置座標や，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ってしまう．

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リアルタイムにデバッグを行える機構を，ゲームシステム記述言語 kameTL 及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もつ 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．以下にリアルタイムデバッグ機構の特徴を示す．

- ソースコード中の実行位置，変数の中身をリアルタイムに表示.
- プログラムの実行していった跡を色分けしてリアルタイムに表示.

結果として，ゲームプレイを中断することなくプログラム内部状態が確認可能になり，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られるようになる．本稿では提案手法，およびその手法を実装したシステムの詳細を述べる．

2 章では、kameTL について説明し、3 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。そして 4 章で提案機構の特徴及び、動作例を紹介し、5 章で提案機構の実現方法を説明する。6 章では関連研究を紹介し、7 章で本稿のまとめを述べる。

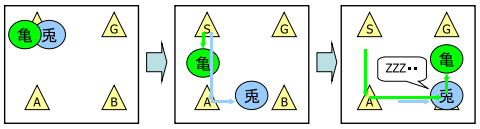

図 1　亀と兎の競争

2　kameTL

ゲームプログラムでは一定間隔ごとに動作の結果を画面に表示しなくてはならない．この時間の間隔を以下，フレームと呼ぶ．1 フレームで，全てのキャラクタを少しずつ動かし画面に表示する，という処理を繰り返すことにより，ゲームを見ている者に，あたかも画面内でたくさんのキャラクタが実時間で動いているように見せることができる．よって，各キャラクタの連続した動作の処理でも，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．スレッドを用いた並行処理の記述例として，図 1 のように亀と兎が競争するということを考えてみる．亀は歩きながら，兎は走りながらA地点，B地点を経由してG地点まで動作させる．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 2 のように書ける．スレッドの処理権限切り替えには明示的に yield 文を用いる．そして，各スレッドは 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．

```
//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 2　亀と兎の競争:kameTL ソースコード

3　ゲームプログラムのデバッグ

ゲームプログラムのデバッグ方法には，統合開発環境付属のデバッガを用いる方法や，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などがある．ここでは，これらの方法とその問題点について述べる．

3.1　統合開発環境付属のデバッガを用いる方法

Microsoft VisualStudio 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．デバッガを用いることにより，図 3 のようにプログラムを一時停止させて内部の変数の値や，ソースコード中のどの位置を実行しているかといった情報が得られる．

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るが，どのような実行経路でそこに至ったかまではわからない．

例えば，プレイヤー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するというプログラムをデバッグすることについて考えてみる．図 4 のように，`SetAnimation("歩く")` が実行される部分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，`v.X == 0.0 && v.Y == 0.0` が偽であったことしかわからず，`KeyDown(VK_S)` 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情報は

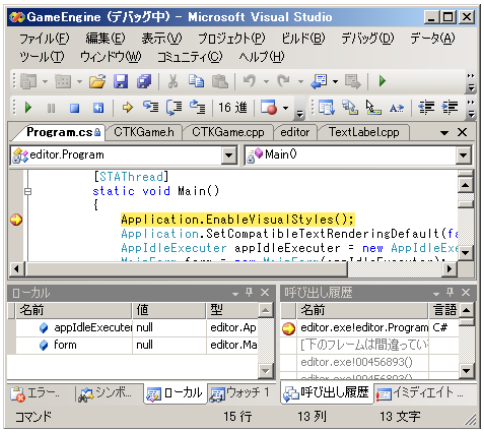

図 3　Microsoft Visual Studio 付属のデバッガ

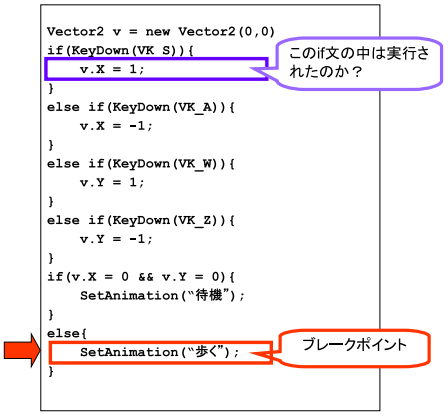

図 4　停止時点の情報しか得られない

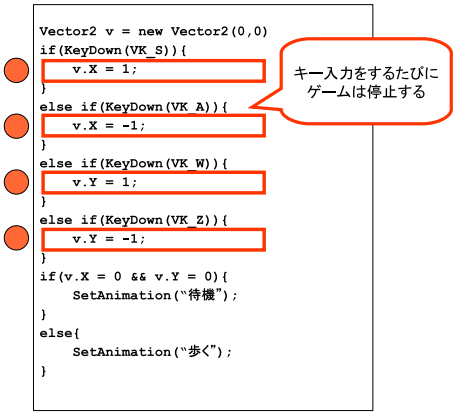

図 5　キー入力のたびにゲームプログラムが停止

わからない．

キー入力が正確に行われているか調べるため，図 4 を変更し，図 5 のように多数のブレークポイントを設置してしまうと，キー入力が行われるたびにゲームが停止してしまい，デバッグ効率が悪くなってしまう．特に，キー入力のタイミングや順番が重要である箇所では，プログラムが一時停止してしまうことによりゲームプログラムが正常に動作しなくなってしまう可能性がある．例えば，格闘ゲームで，”自分が操作するキャラクタが敵キャラクタに十分近いときに，A キーを押し，その 0.5 秒に B キーを押し更に 0.5 秒後に両方のボタンを離すと技が発動する”といった処理が正しく動いているかデバッグする場合，ゲームプログラムが途中で停止してしまうと正常な入力が行えなくなってしまう．

また，キャラクタの位置座標等のゲームプレイ中に刻々と変化していく内部変数の情報を，リアルタイムにゲームプレイを中断せず観察することができないことも問題である．

3.2　必要な情報を画面に出力する方法

ゲームプログラムをデバッグするもうひとつの方法として，必要な情報を画面に出力するという方法がある．例えば，図 6 では，キャラクタの速度変数 v の値と，どのキーが押されたかの情報をゲーム画面上に表示している．この方法では，ゲームプレイを中断することなく，1 フレームごとに変化していく変数の値や，プログラム中の特定の箇所が実行されたかという情報をリアルタイムに観察することが可能である．しかし，この方法では図 7 のように，プログラムのあちこちに `ShowScreen(...)` とデバッグ情報表示用のコードを記述しなければならず，プログラムが汚くなってしまう上に，開発者にとっては面倒な作業である．

4　リアルタイムデバッグ機構

このような問題点を解決するため，ゲームプレイを停止させることなくリアルタイムにデバッグを行える機構を提案する．リアルタイムデバッグ機構の特徴と動作例について述べる．

4.1　特徴

リアルタイムデバッグ機構の主な特徴は次のとおりである．

- ソースコード中の実行位置，変数の値をリアルタイムに表示．

図 6　キー入力のたびにゲームプログラムが停止

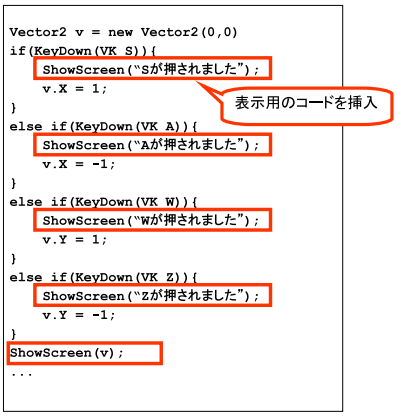

図 7　デバッグ情報表示用のコードを挿入

- プログラムの実行していった跡を色分けしてリアルタイムに表示.

ソースコード中の実行位置や変数の値の情報は，ゲームが進行するにしたがってリアルタイムに更新されていく．このため，開発者はゲームプレイを中断することなく，効率よくデバッグ作業を行うことができる．

また，それぞれのスレッドについて，1 フレームの間に実行したソースコード内の行を緑色で表示する．表示する際には前に実行された行ほど薄く表示されるようになっている．なので，1 フレームの間にプログラムが実行していった跡が，透明，薄緑色，緑色とグラデーションしながら表示される．また，実行された行についた色は 1 秒間かけてフェードアウトしていく．このため，開発者は多数のブレークポイントを設置せずとも，プログラム中のどの行が実行されたのかの情報が得られるようになる．

4.2　動作例

図 8 は，kameTL で記述されたロールプレイングゲームを kameTL 統合開発環境上でデバッグしている様子である．図 8 では，ゲーム内のキャラクタ等を制御するスレッドの中から，兵士のキャラクタの動作を制御しているスレッドを選び，更にそのスレッドの中のメソッドフレームの中から，**ノーマル** メソッドを選び，そのメソッド内でのプログラムの実行の様子をリアルタイムに表示している．

ゲーム画面内で兵士のキャラクタが移動すると，ソースコード中の実行している行の表示もそれに対応し，リアルタイムに更新されていく．このため，開発者はゲームプレイをし続けながらでも，各キャラクタが現在ソースコード中のどの位置を実行しているかを把握できるため，プログラムの動きが理解しやすくなると同時にデバッグの効率も上がる．kameTL では，ひとつのスレッドがひとつのキャラクタの動作と一対一に対応しているため，この手法は特に効果的である．

画面の右下に表示されているのは，兵士キャラクタの位置座標である．これも兵士キャラクタが移動するにしたがってリアルタイムに値が変わっていく．例えば，プレイヤーの操作するキャラクタが兵士キャラクタに近づいたときだけ兵士キャラクタの位置座標が微妙にずれてしまうバグがあったときに兵士キャラクタの位置座標や，関連しそうな変数をリアルタイムに監視しながら，色々な操作を試してみてバグの原因を探すといったことが可能になる．

図 9 は，入力処理を受け付けて，キャラクタを動かすスレッドが 1 フレームの間に実行した後の色分け表示である．プレイヤーは，自身の操作しているキャラクタを動かすために Z キーと S キーを押しており，それに対応してプログラム内では，`System.keyDown(VK_Z)` と，`System.keyDown(VK_S)` が真になり，それぞれ if 文の中が実行されていることが確認できる．このように，多数のブレークポイントを設置しなくても多くの情報が得られるようになる．

5　実装

リアルタイムデバッグ機構は kameTL 及び，その統合開発環境上に実装した．本システム全体の構成を図 10 に示す．

kameTL はコンパイラと仮想機械から構成される．

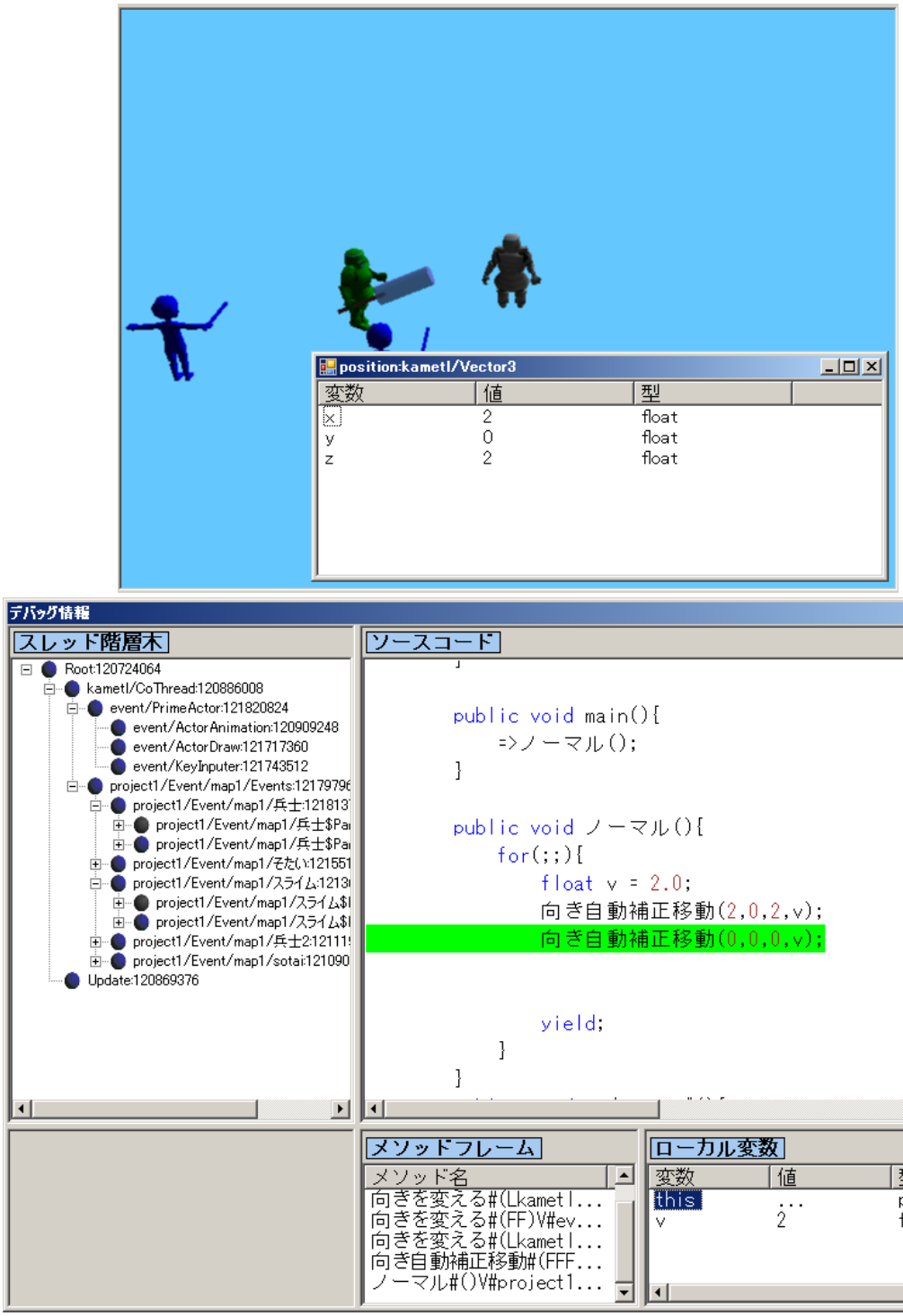

図 8　ソースコード中の実行位置，変数の値をリアルタイムに表示

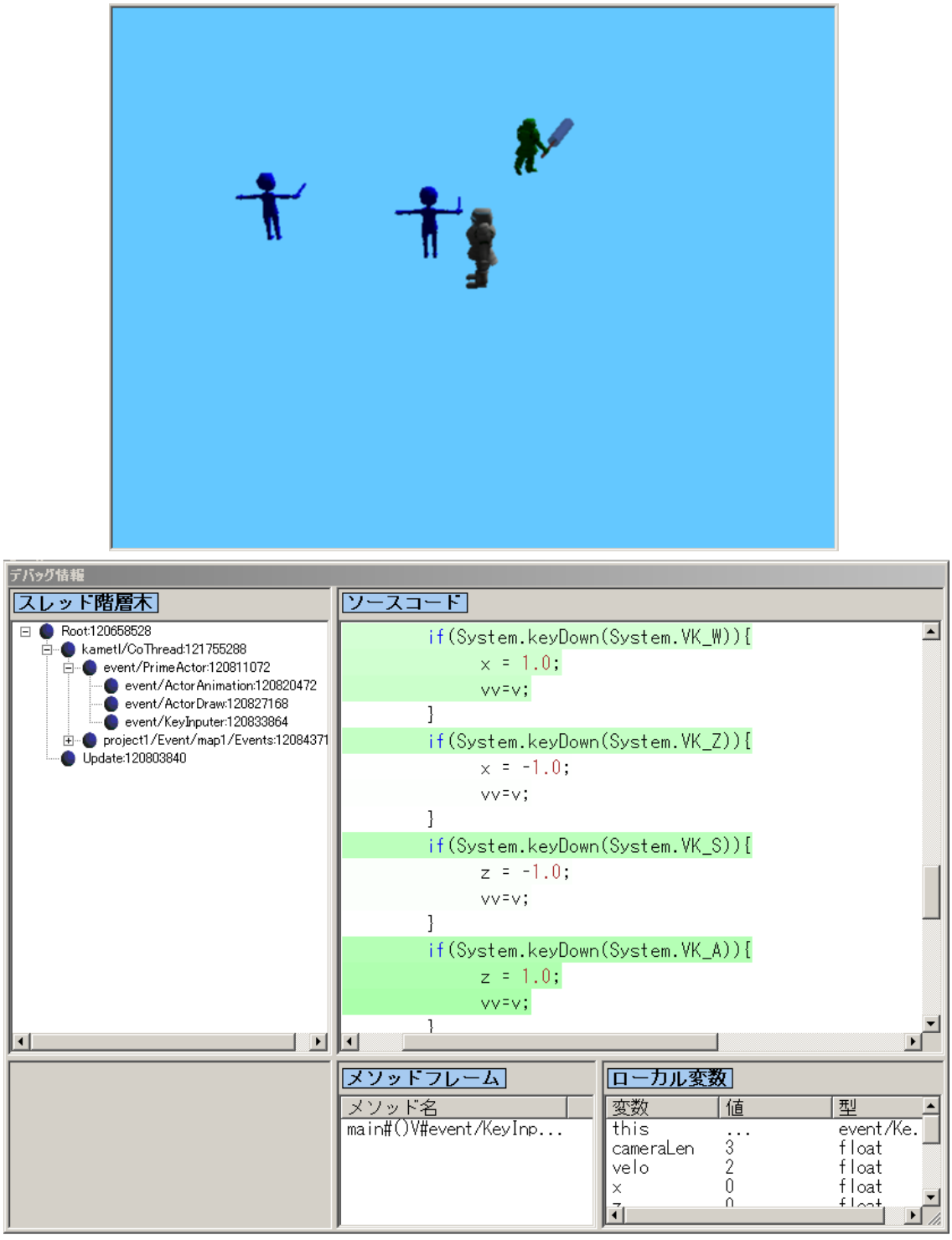

図 9　プログラムの実行していった跡を色分けしてリアルタイムに表示

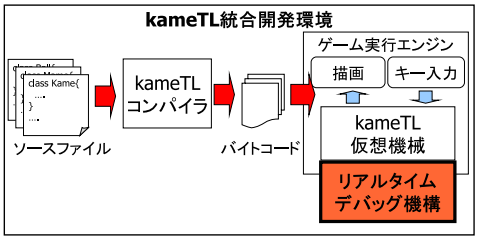

図 10　システム全体の構成

コンパイラはソースコードをバイトコードへと変換し，仮想機械がそれを読み込み実行する．kameTL の仮想機械は Java の仮想機械に似たスタックマシンであり，C++ で記述されている．並行処理のための一部の命令などを除きバイトコードの仕様もほぼ Java と同じである．また，統合開発環境やデバッガのための GUI 部分は C# を用いて開発した．

5.1　kameTL 仮想機械の詳細

仮想機械のスレッド制御及びバイトコード実行部分の擬似コードを図 11 に示す．ExecuteOneFrameStep メソッドは 1 フレームの間に，各スレッドを 1 度ずつ実行していき，最後にゲーム画面の更新を行う．各スレッドはそれぞれ ExecutebyteCode 関数を呼び出し，関数内でバイトコードの命令を次々に実行していく．

例えば istore0 命令は，メソッドフレームスタックの最上位にあるメソッド，つまりスレッドが現在実行中のメソッドの 0 番目の int 型ローカル変数の値をスタックにプッシュする．iadd 命令はスタックから 2 つの int 型の値をポップして取り出し，それらを加算した結果をスタックにプッシュする．yield 命令はスレッドの切り替えのためのものであり，この命令に到達すると，ByteCodeExecute 関数から戻り，次のスレッドへと処理が移る．

5.2　リアルタイムデバッガ実現のための手法

リアルタイムにソースコード中の実行位置，そして変数の値を更新するためには，どのようなタイミングで情報を更新すれば，開発者にとって有益な情報となるか，そしてどのような頻度で更新を行えばオーバーヘッドを抑えられるかが問題となる．ここでは，ゲームプログラム特有の性質を利用し，このタイミングと頻度の問題を一挙に解決する．

通常ゲームプログラムは 1 フレームの間に，プレイヤーの入力を受け取りキャラクタなどを少しずつ動かし，最後にその結果を画面に反映させる，という処理の繰り返しである．そのため，キャラクタの位置座標等の値は 1 フレームを境に変更されることが多い．また，入力処理ではプレイヤーの全入力を調べ，その入力にしたがってキャラクタを動かすといった処理を毎フレームごとに行っている．つまり，ゲームプログラムでは 1 フレームという処理の切れ

```
//
//  1 フレームで全てのスレッドを一度ずつ実行する
//
void ExecuteOneFrameStep(){
    //スレッドの実行とゲーム画面の更新を繰り返す
    for(;;){
        //全スレッドを 1 つずつ実行する
        for(int i=0;i<スレッド数;i++){
            RESULT result =
              cothreads[i].ExecuteByteCode(...);
            ...
        }
        //画面の更新等を行う
        UodateDisplay(...);
    }
}

//
//  バイトコード解釈，実行部分
//
RESULT CoThread::ExecuteByteCode(...){
    ...
    ...
    istore0
    int a = 0 番目のローカル変数を取得 ();
        スタックにプッシュ (a);
        goto code[pc++];//次の命令へジャンプ
    ...
    ...
    iadd:
        int a = スタックからポップ ();
        int b = スタックからポップ ();
        スタックにプッシュ (a+b);
        goto code[pc++];//次の命令へジャンプ
    ...
    ...
    yield:
        //このスレッドを中断する
        return RESULT_YIELD;
        goto code[pc++];//次の命令へジャンプ
    ...
    ...
}
```

図 11　スレッド制御及び，バイトコード実行部分の擬似コード

目で情報を更新すると，開発者にとっては有益な情報が得られるといえる．

バイトコードの記録は，図 11 の ExecuteByteCode メソッドにおいて，各命令の最後の goto code[pc++]; の前に**バイトコードを記録（pc，現在のメソッド名）;** というコードを挿入するだけで行える．1 フレームの間に実行したバイトコードを記録しておくのはデバッグ対象のスレッドに関してのみ行えばよいので，たいしたオーバーヘッドではない．

6 関連研究

Haskell Program Coverage [1] 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一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研究と似ているが，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ことはできない．

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 [2] 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，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をもっているが，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．また，Java はゲームに適した並行処理機構をもっておらず，kameTL のようにひとつのキャラクタの一連の処理の流れを，ひとつのスレッドに関連付けることができないため，本機構のように，キャラクタの処理の流れをわかりやすく表示することもできない．

アクションゲーム記述に特化した言語 [3]，Tonyu System[4] 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．アクションゲーム記述に特化した言語 [3] は，デバッグ機構を備えていない．Tonyu System[4] は，デバッグ機構として変数の値をリアルタイムに監視する機能はもつが，ソースコードの実行中の位置をリアルタイムに表示する機能はない．

7 結論

本稿ではゲームシステム記述言語 kameTL 上に，リアルタイムデバッグ機構を実装しその効果を示した．リアルタイムデバッガを用いることにより，開発者はゲームプレイを中断せず，リアルタイムにプログラム内の実行位置や，変数の値を観察することが可能になり，デバッグ効率が大きく上がると期待できる．

今後の課題としては，スレッド一覧の表示，ローカル変数の表示をリアルタイムに行えるようにすること，そして複数のスレッドの処理を同時に観察できるようにすることなどがあげられる．また，現在では for 文の中など 1 フレームの間に何度も通った箇所は，最後に通った跡が優先して表示されるようになっているが，このような何度も通った箇所をわかりやすく表示する工夫についても考える必要がある．

謝辞

本機構を開発するにあたり，終始手厚い御指導をして下さった岩崎英哉教授，大山恵弘准教授，鈴木貢先生に感謝いたします．また，本機構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”の一部として開発しました．ご支援いただいております IPA，竹内郁雄教授，株式会社 オープンテクノロジーズの皆様，共同開発者の唐澤君，川ノ上君に厚くお礼申し上げます．

参考文献

[1] Andy Gill et al.: Haskell Program Coverage. Haskell'07. p1-12 (2007)

[2] 佐藤 ら: 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)

[3] 西森ら: アクションゲーム記述に特化した言語．情報処理学会誌 Vol.44 No.SIG15 p36-54 (2003)

[4] Tonyu System: http://tonyu.jp/