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

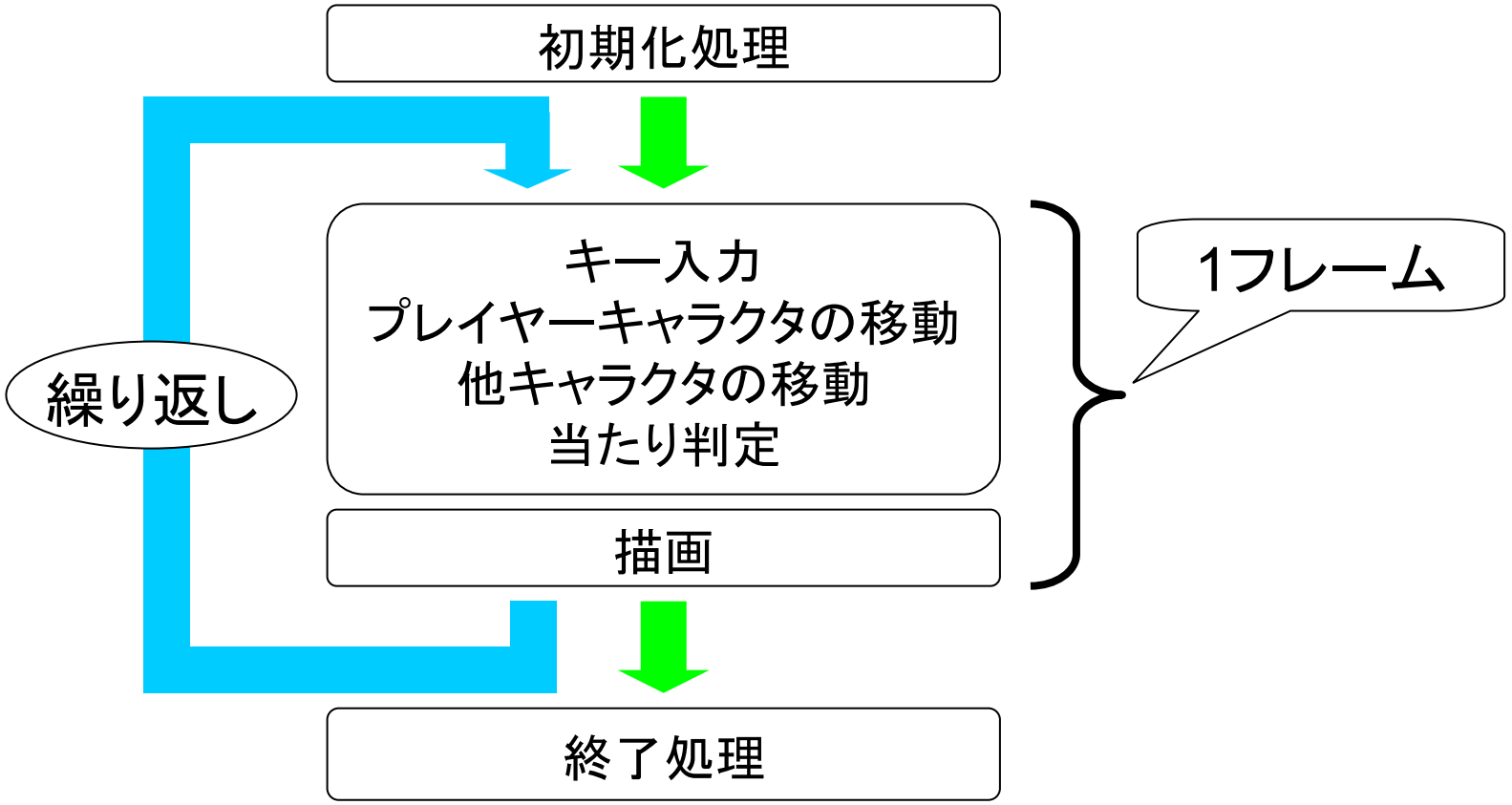
2008年3月17日（月）

電気通信大学
電気通信学研究科 情報工学専攻
丹野 治門

発表の流れ

- 既存のゲームデバッグ方法における問題点
  - プログラムを中断させるデバッグ方法は向いていない
- ゲームプログラムに適したデバッガ
  - プログラムを動かし続けながらデバッグができる
  - ゲームシステム記述言語kameTLで実現した
- 活用事例
- 実装
- オーバーヘッド
- 関連研究
- まとめ

ゲームプログラムの動作



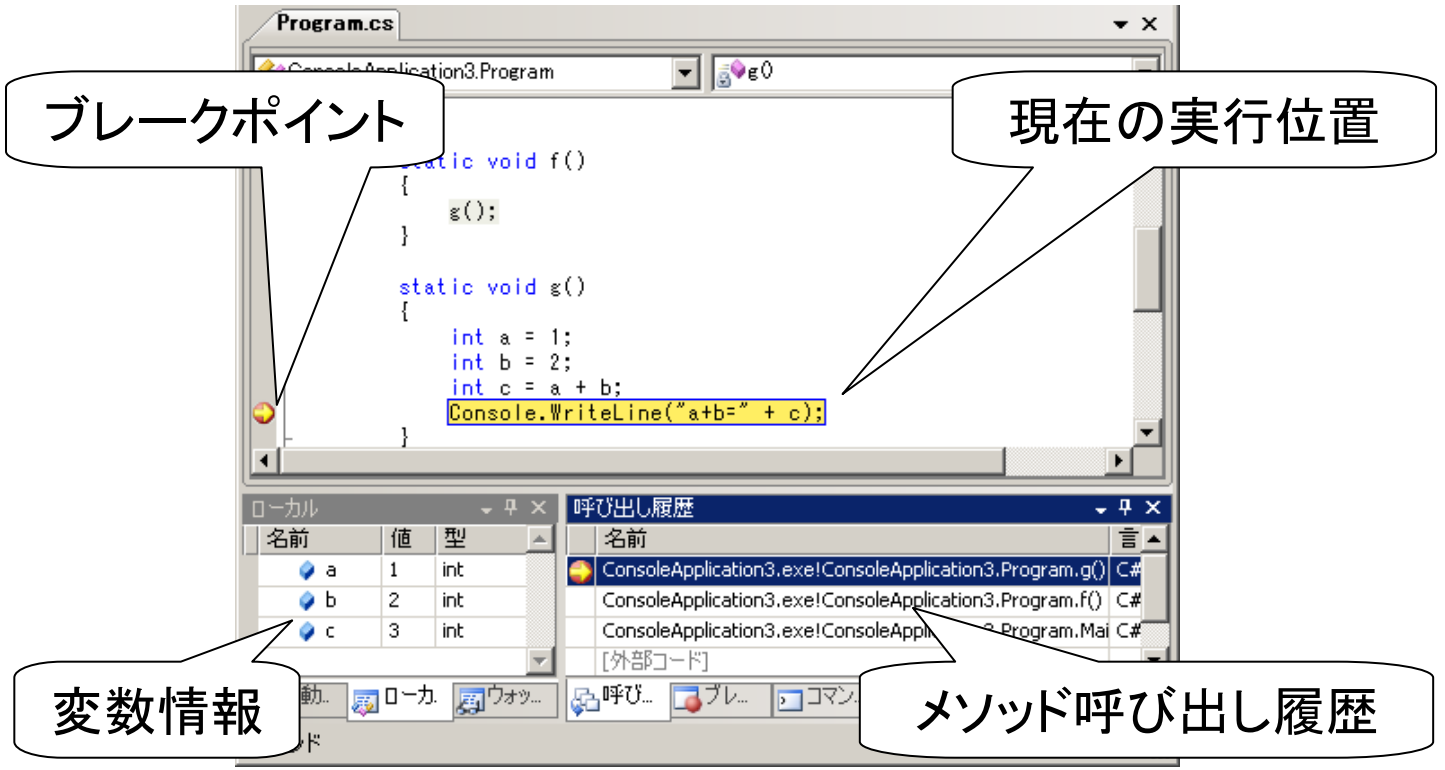
- フレームごとにプログラム内部の状態が変わっていく
- 繰り返し部分がプログラムの大半を占める

ゲームプログラムのデバッグ

- デバッグ対象の大半は繰り返し部分
- 繰り返し部分の処理
  - キー入力等でプログラム内部の状態が変わる
    ➔インタラクティブ性が高い
  - 毎秒数十回の速さで繰り返される
    ➔リアルタイム性が高い

- 従来のデバッグ方法
  - 統合開発環境付属のデバッガを用いる
  - 画面に必要な情報を表示する

デバッガを用いる方法



Microsoft Visual Studio 2008

- プログラムを中断し、プログラム内部状態を観察できる
- インタラクティブ性の高いゲームプログラムには向かない

デバッガを用いる方法の問題点①

- プログラム停止時点の情報しか得られない
  - どのような実行経路でそこに至ったかはわからない

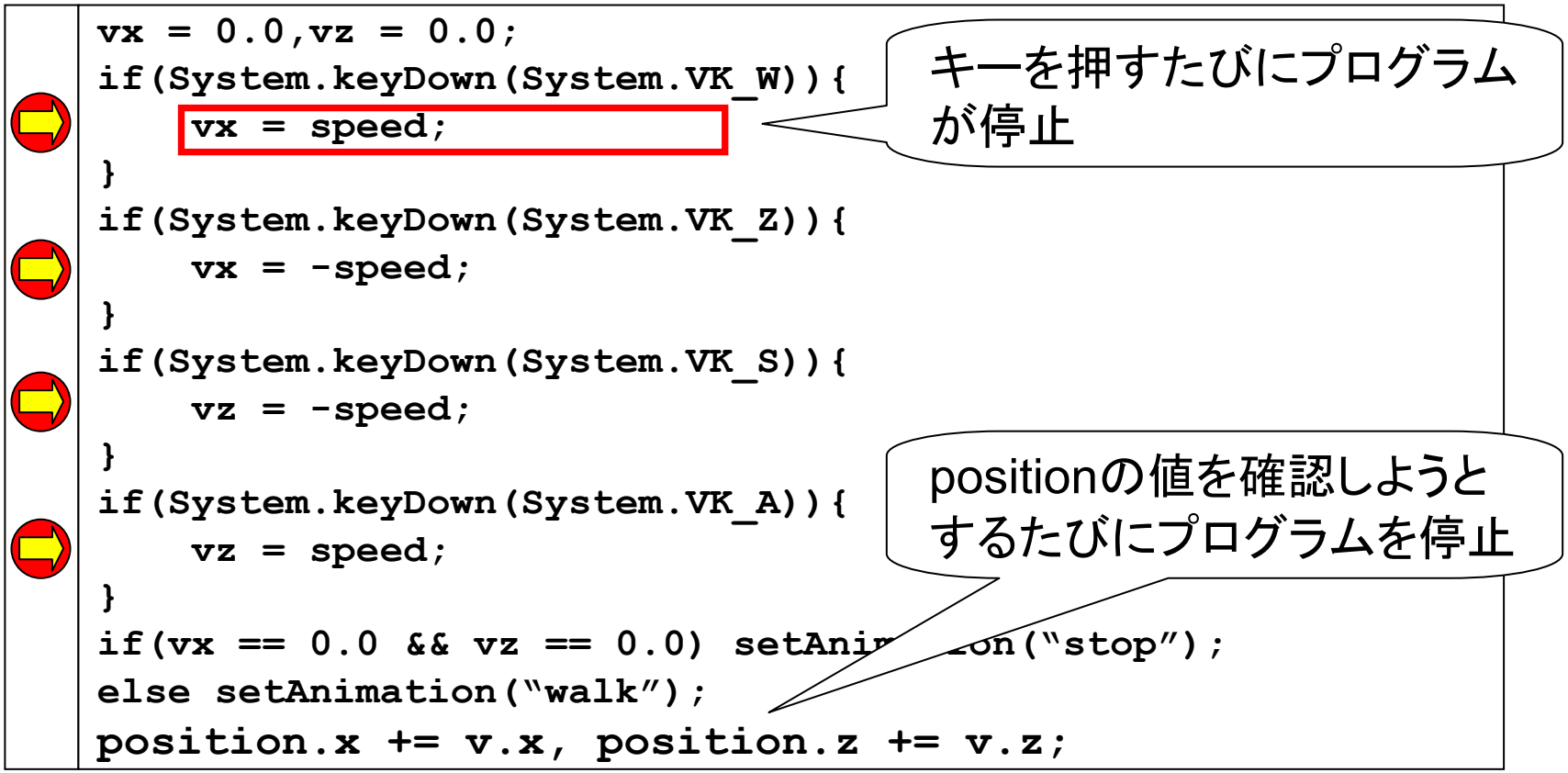
```
vx = 0.0,vz = 0.0;
if(System.keyDown(System.VK_W)){
    vx = speed;
}
if(System.keyDown(System.VK_Z)){
    vx = -speed;
}
if(System.keyDown(System.VK_S)){
    vz = -speed;
}
if(System.keyDown(System.VK_A)){
    vz = speed;
}
if(vx == 0.0 && vz == 0.0) setAnimation("stop");
else setAnimation("walk");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

このif文の中は実行されたのかわからない

プログラムの実行位置

デバッガを用いる方法の問題点②

- デバッグ時にゲームプレイを中断する必要がある
  - デバッグ効率が悪い
  - タイミングが要求される個所では、期待通りに動作させられなくなる



画面に出力する方法

- リアルタイムにプログラム内部の状態を観察できる
- デバッグ用のコードを埋め込む必要がある

画面に出力する方法の問題点

```
float vx = 0.0,vz = 0.0;
float speed = getDeltaTime()*4.0;
if(System.keyDown(System.VK_W)){
    vx = speed;
    display("keyboard input W");
}
…
else if(System.keyDown(System.VK_A)){
    vz = speed;
    display("keyboard input A");
}
if(vx == 0.0 && vz == 0.0) setAnimation("stop");
else setAnimation("walk");
position.x += v.x, position.z += v.z;
display(position.toString());
```

実行経路情報を表示

変数情報を表示

- ソースコードの可読性が下がる
- プログラマの負担が大きい

問題点のまとめ

- 従来型デバッガの問題点
  - プログラムを中断しないとデバッグ情報を観察できない

  ➔デバッグ効率が悪くなる

  ➔タイミングが重要な個所で期待通りに動作させられない

- 画面に出力する方法の問題点
  - デバッグ用のコードを埋め込む必要がある

  ➔ソースコードの可読性が下がる

  ➔プログラマの負担が大きい

本研究の目的

- ゲームプログラムに適したデバッガを提案
  - プログラムを中断せずにデバッグ情報を観察可能にする

  ➔デバッグ効率を良くする

  ➔プレイヤーの入力を受け付けながらデバッグ可能にする

- 対象とするゲーム
  - 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
  - ネットワークゲーム

ゲームプログラムに適したデバッガ

- 設計方針
  - ゲーム内の情報はフレームを境に変更される性質を利用

  ➔実行経路情報、変数情報をフレームごとに更新、表示

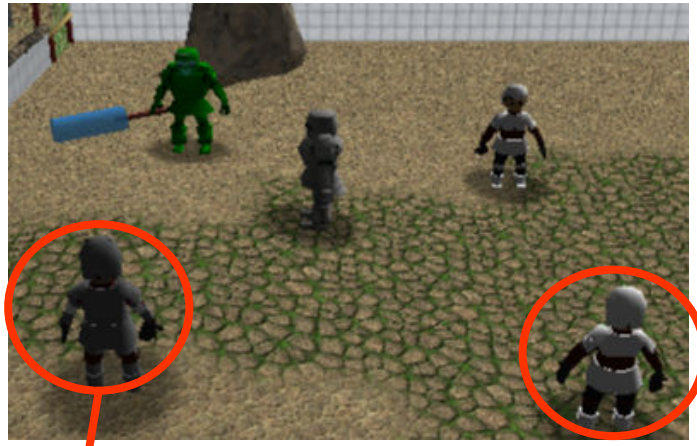
- 実行経路情報表示の手法
  - 時間的に後に実行された行ほど濃くグラデーション表示

  ➔視覚的にプログラムの実行経路を把握しやすくする

- kameTL[丹野, 08]上に実装
  - ゲームに特化した並行処理機構をもつ
  - Java風のオブジェクト指向言語

kameTL

- ゲームに適したノンプリエンプティブスレッド機構をもつ



```
public void 移動(Vector3 to,int v){
  while(!pos.equals(to)){
    updateVector3(pos,to,v);
    yield;
  }
}
```

スレッドの切り替え

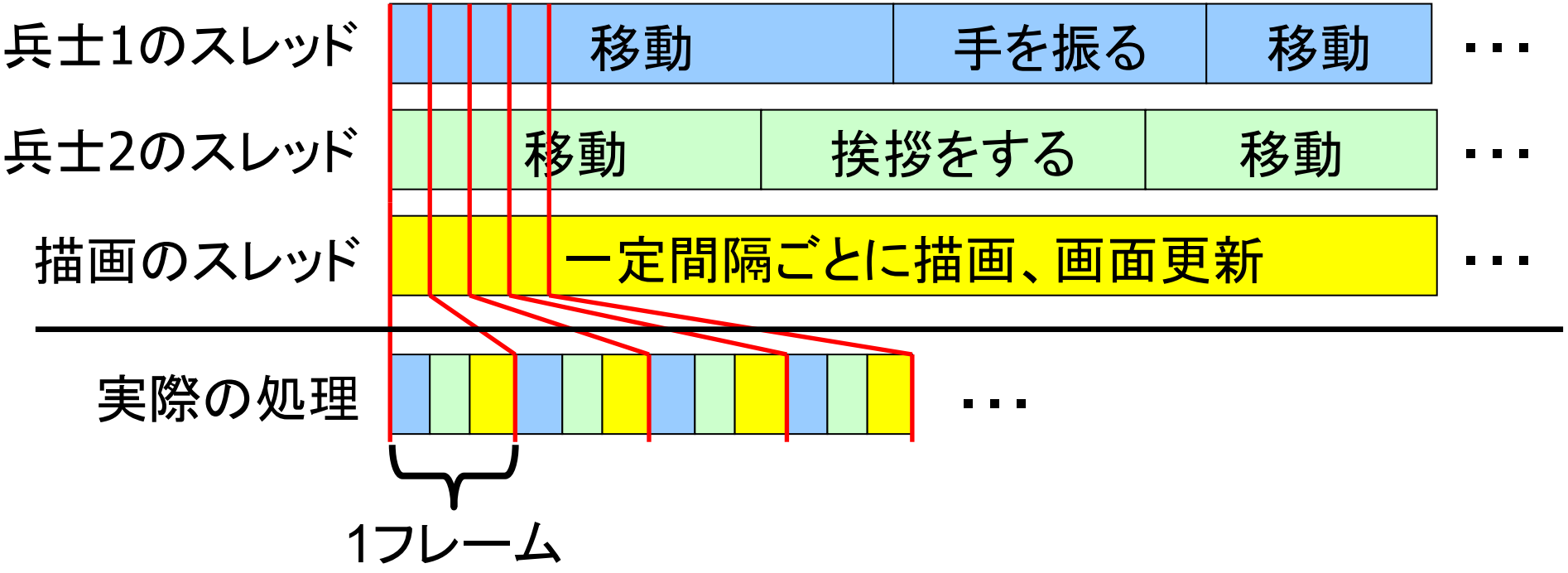
```
class 兵士1 extends CoThread{
  public void main(){
    移動(new Vector3(0,0,5),8);
    アニメーション("手を振る");
    移動(new Vector3(0,0,7),8);
  }
}
```

```
class 兵士2 extends CoThread{
  public void main(){
    移動(new Vector3(2,0,5),2);
    アニメーション("挨拶をする");
    移動(new Vector3(4,0,7),2);
  }
}
```

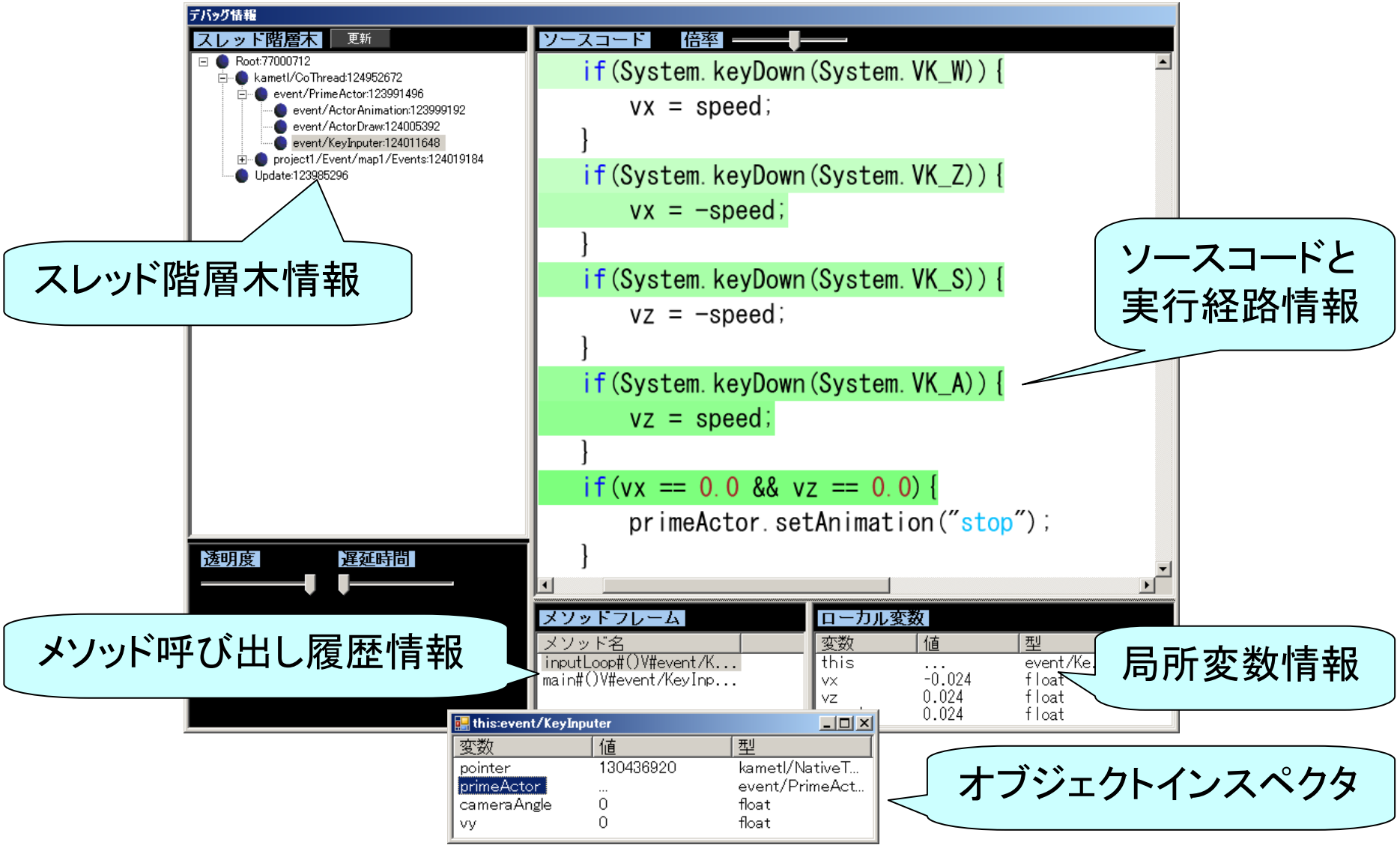
キャラクタの動作の流れを自然に記述

kameTL：スレッド

- ゲームに適したノンプリエンプティブスレッド機構
  - 明示的なコンテキスト切り替え（ノンプリエンプティブ）
  - スレッドには親子関係があり、スレッド階層木を形成する
  - 1フレームごとに全スレッドを1度ずつ深さ優先順で実行する



デバッガの画面構成


デバッグ情報
スレッド階層木
更新
Root:77000712
kametl/CoThread:124952672
event/PrimeActor:123991496
event/ActorAnimation:123999192
event/ActorDraw:124005392
event/KeyInputer:124011648
project1/Event/map1/Events:124019184
Update:123985296
ソースコード
倍率
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");
}
透明度
遅延時間
メソッドフレーム
メソッド名
inputLoop#()V#event/K...
main#()V#event/KeyInp...
ローカル変数
変数
値
型
this ... event/Ke...
vx -0.024 float
vz 0.024 float
0.024 float
this:event/KeyInputer
変数
値
型
pointer 130436920 kametl/NativeT...
primeActor ... event/PrimeAct...
cameraAngle 0 float
vy 0 float
スレッド階層木情報
ソースコードと
実行経路情報
メソッド呼び出し履歴情報
局所変数情報
オブジェクトインスペクタ

提案デバッガのデモ

- 機能紹介
- 活用事例紹介

提案デバッガの特徴

- リアルタイムにプログラムの状態を観察可能
  - プログラムの実行経路情報
  - プログラム内部の変数情報

➔ゲームプレイを中断せず効率よくデバッグできる

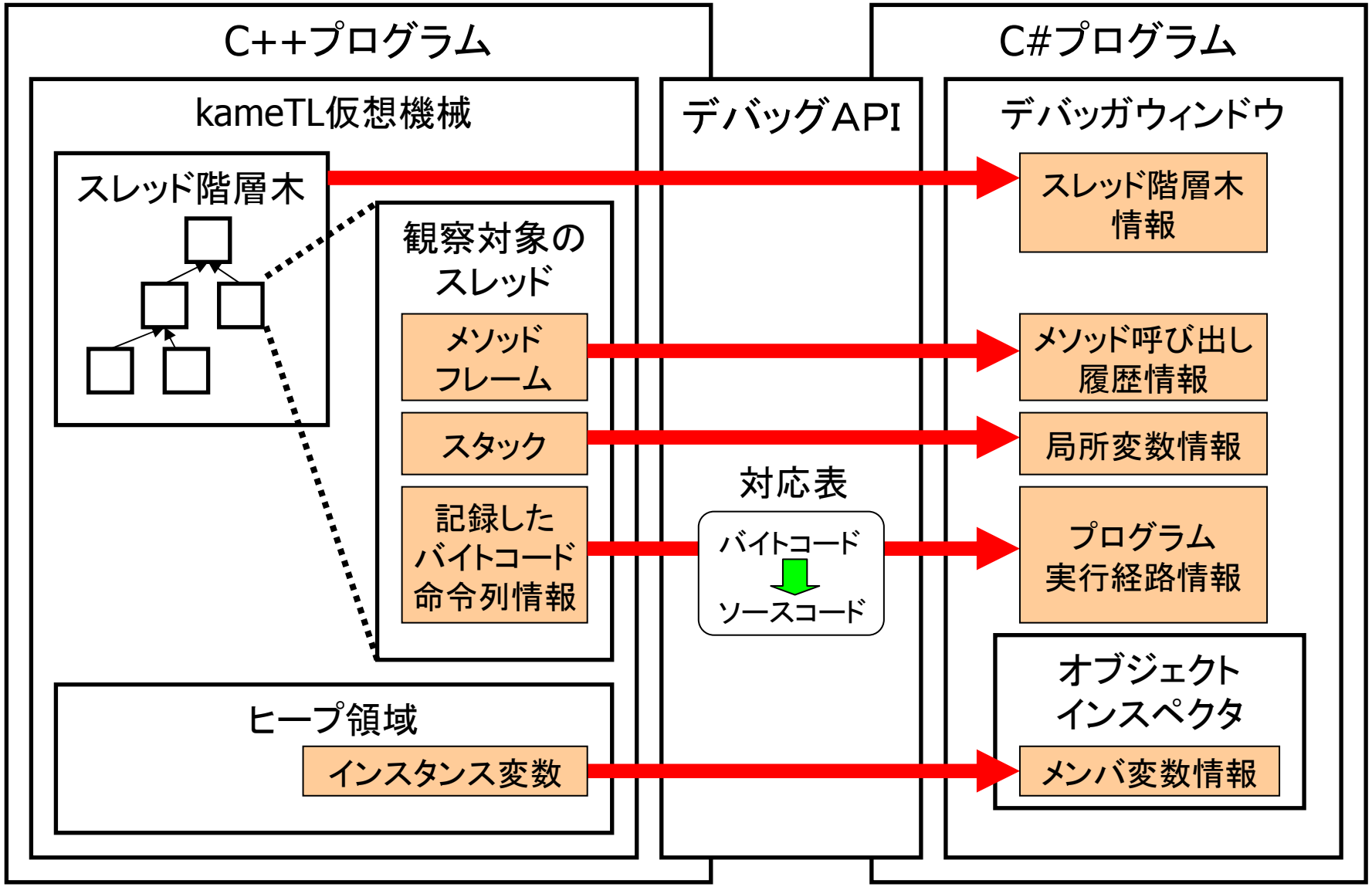
- プログラムの実行経路情報をグラデーション表示
  - 色を変えていくことにより、実行の跡をわかりやすく表示

➔多数のブレークポイントを配置せず、多くの有用な情報を得られる

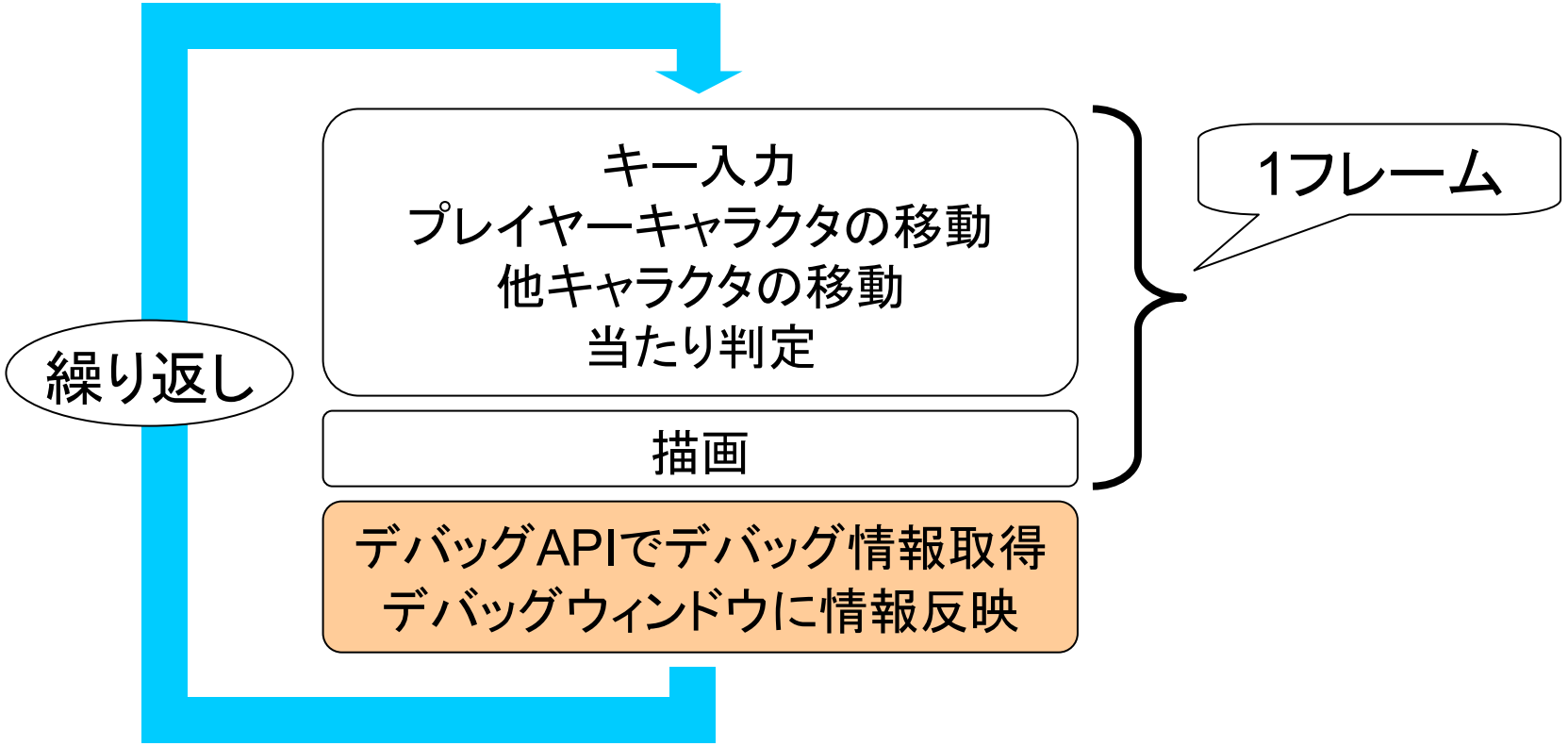
実装

- kameTLコンパイラ
  - kameTLのソースコードをバイトコードへ変換
  - デバッグ情報をバイトコードに付加
- kameTL仮想機械（Java仮想機械に類似したスタックマシン）
  - バイトコードを読み込み、プログラムを実行
  - 観察対象のスレッドは1フレーム間に実行した命令列を記録
  - デバッグAPIを通して、各種デバッグ情報取得可能

デバッガの構成


C++プログラム
kameTL仮想機械
スレッド階層木
観察対象の
スレッド
メソッド
フレーム
スタック
記録した
バイトコード
命令列情報
ヒープ領域
インスタンス変数
デバッグAPI
対応表
バイトコード
ソースコード
C#プログラム
デバッガウィンドウ
スレッド階層木
情報
メソッド呼び出し
履歴情報
局所変数情報
プログラム
実行経路情報
オブジェクト
インスペクタ
メンバ変数情報

デバッガの動作



- デバッグ情報はフレームとフレームの間に取得

➔全スレッドは停止しているので、安全に情報を取得できる

オーバーヘッド

- 実験環境
  - CPU…Intel Core2 Duo 3.0GHz、Mem…2GB、
    ビデオカード…GeForce8600GT、OS…WindowsXP SP2
- シューティングゲームで1フレームの平均処理時間を計測
  - デバッガを使用する場合としない場合
  - 入力処理用スレッドを半透明表示したデバッグウィンドウで観察

敵キャラクタの数	800	1200	1600	2000
デバッガ無し(ms)	17.08	25.06	32.18	39.52
デバッガ有り(ms)	25.09	33.55	40.04	47.06
オーバーヘッド(%)	46.9	33.9	24.4	19.0

- スレッドが増えると相対的にオーバーヘッドは小さくなる

関連研究①

- Haskell Program Coverage [Gillら, 07]
  - プログラムが評価した箇所を色分けして表示する
    - 毎回真だった箇所を緑、毎回偽だった箇所を赤
  - プログラム実行中にリアルタイムに表示はしない

- 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 [佐藤ら, 07]
  - マウスクリックなどに対応じて、ソースコード中の実行位置を強調表示する
  - 実行経路情報をグラデーション表示する機能はない
  - 変数をリアルタイムに観察する機能もない

関連研究②

- ゲームに特化した並行処理機構をもつ言語
  - アクションゲーム記述に特化した言語 [西森ら, 03]
    - デバッグ機構をもたない
  - 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[長, 01]
    - 変数情報をリアルタイムに観察することができる
    - 実行位置情報をリアルタイムに表示できない

まとめ

- ゲームプログラムに適したデバッガを提案
  - 実行経路情報、変数情報を1フレームごとに更新する
  - ゲームプレイを継続しながらデバッグできる
  - 基本的な仕組みは他の言語にも応用可能と考えられる

- 今後の課題
  - 本デバッガが広範囲のゲームで有効なことを確認する
    - ネットワークゲーム