プログラムを停止させないデバッグを可能とする手法の提案

丹野 治門，岩崎 英哉（電気通信大学）

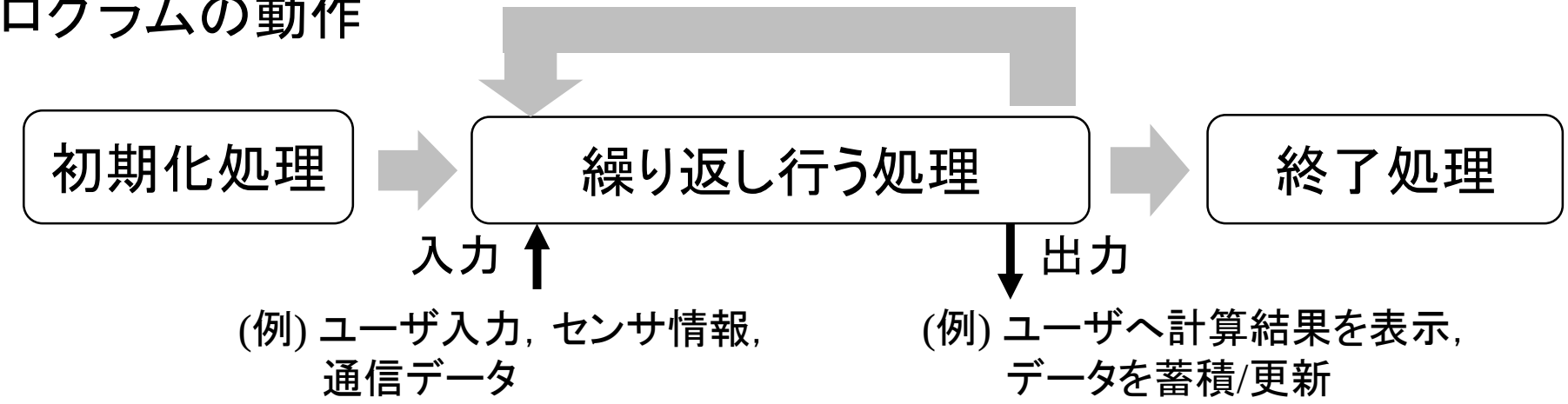
2018年8月31日

目次

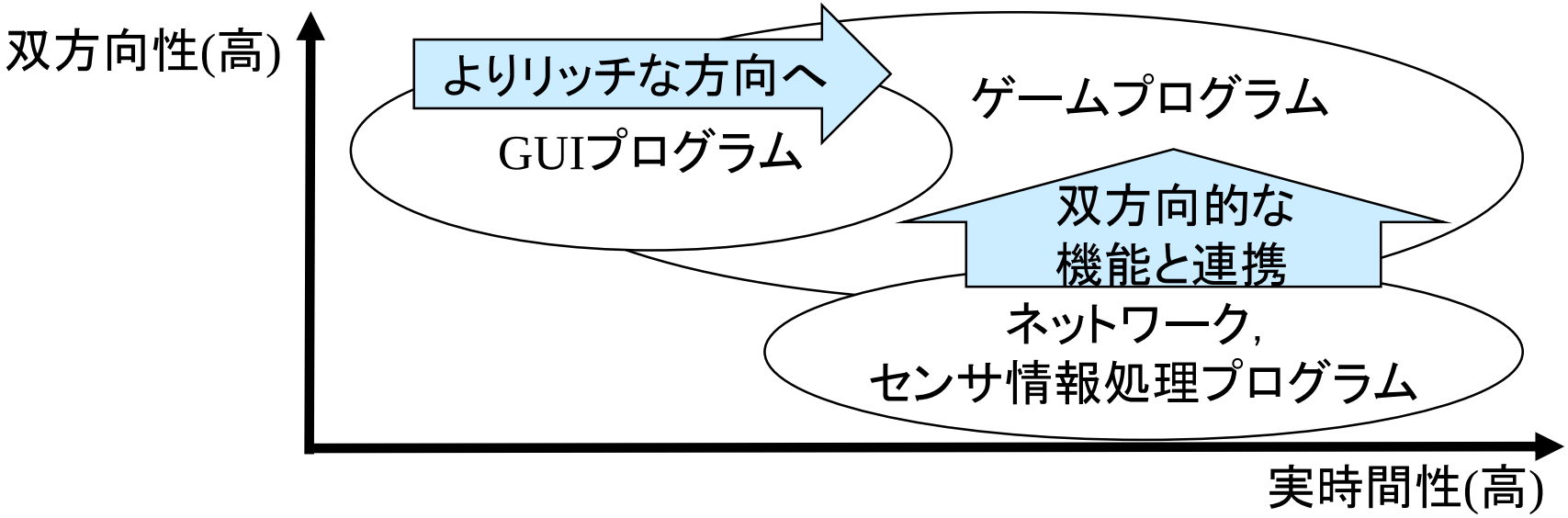
- 双方向性，実時間性の高いプログラムのデバッグの問題点
  - 例：ゲーム，GUI，ネットワーク，センサ情報処理プログラム
  - プログラムを停止させる従来デバッガは適さない
- プログラムを停止させないデバッグ手法を提案
  - 機能，活用事例のデモ
- デバッガの実現方式（C#）
- オーバヘッド計測
- 関連研究
- まとめ

双方向性，実時間性が高いプログラム

プログラムの動作



様々なプログラムに高い双方向性，実時間性が求められる



双方向性，実時間性が高いプログラムのデバッグ

- デバッグ対象の大半は「繰り返し行う処理」

- 従来型のデバッグ手法
  - ブレークポイントでプログラムを停止し，内部状態を観察

  ➔双方向性，実時間性の高いプログラムのデバッグでは，プログラムを停止させる手法は適さない
    - デバッグ効率が悪い
    - タイミングが要求される箇所では，期待通りに動作させられなくなる

(例)アクションゲーム

アクションゲームでプレイヤが敵キャラクタへ攻撃を行うロジック

```
private void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
    int damagePoint = player.OffensivePower;
    if (input.AttackButton){//プレイヤが攻撃中
        if (input.DashButton) damagePoint *= 5; //ダッシュ中はダメージ5倍
            foreach (var enemy in enemyList){
                if (Vector3.Distance(player.Position, enemy.Position) <= 5.0){
                //敵がプレイヤーの近距離に存在する
                    if (!enemy.IsInvincible){
                    //敵が無敵状態ではない
                        enemy.HitPoint -= damagePoint; //敵のHPを減らす
                        if (enemy.HitPoint <= 0) enemy.Dead(); //敵はHPが0以下で消滅
                    }
                    enemy.EndPlayerCollision();
                }
            }
        }
    }
}
```

- ユーザの入力により挙動が変化してゲームが進む➔双方向性(高)
- 処理は一定間隔(例えば1 秒間に30 回)➔実時間性(高)

(例)アクションゲームのデバッグ

アクションゲームでプレイヤが敵キャラクタへ攻撃を行うロジック

```
private void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
    int damagePoint = player.OffensivePower;
    if (input.AttackButton){//プレイヤが攻撃中
        if (input.DashButton) damagePoint *= 5; //ダッシュ中はダメージ5倍
            foreach (var enemy in enemyList){
                if (Vector3.Distance(player.Position, enemy.Position) <= 5.0){
                //敵がプレイヤーの近距離に存在する
                    if (!enemy.IsInvincible){
                    //敵が無敵状態ではない
                        enemy.HitPoint -= damagePoint; //敵のHPを減らす
                        if (enemy.HitPoint <= 0) enemy.Dead(); //敵はHPが0以下で消滅
                    }
                    enemy.EndPlayerCollision();
                }
}
```

＜バグ＞
プレイヤがダッシュボタンを押し，5 倍の攻撃力で敵キャラクタを攻撃すると2-3回で倒せるはずだが，倒せない．
➔このバグの原因をどのように特定するか？

(例)アクションゲームのデバッグ

攻撃ボタンを押したがinput.AttackButton が真になっていない？

input.DashButtonが真になっておらず ダメージが5倍になっていない？

EnemyListに攻撃対象の 敵キャラクタが含まれていない？

Vector3.Distanceの 計算結果が正しくない？

enemy.IsInvincible が (想定と異なり) 真になっている？

```
private vo          rAttack(PlayerInput input, P
    int damageP     t = player.OffensivePower;
    if (input.AttackButton){//プレイヤーが攻撃中
        if (input.DashButton) damagePoint *= 5; //ダッシュ中はダメージ5倍
            foreach (var enemy in enemyList){
                if (Vector3.Distance(player.Positi
                //敵がプレイヤーの近距離に存在する
                if (!enemy.IsInvincible){
                //敵が無敵状態ではない
                    enemy.HitPoint -= damagePoint; //敵のHPを減らす
                    if (enemy.HitPoint <= 0) enemy.Dead(); //敵はHPが0以下で消滅
                }
                enemy.EndPlayerCollisio
            }
        }
    }
}
```

バグの原因は複数考えられる

従来デバッガの問題点

- デバッグ時にプログラムを停止する必要がある
  - デバッグ効率が悪い
  - タイミングが要求される箇所では，期待通りに動作させられなくなる

```
private void PlayerAttack(PlayerInput input, Player player, List<Enemy> enemyList){
    int damagePoint = player.OffensivePower;
    if (input.AttackButton){//プレイヤが攻撃中
        if (input.DashButton) damagePoint *= 5; //ダッ
            foreach (var enemy in enemyList){
                if (Vector3.Distance(player.Position, enemy.Position) <= 5.0){
                //敵がプレイヤーの近距離に存在する
                    if (!enemy.IsInvincible){
                    //敵が無敵状態ではない
                        enemy.HitPoint -= damagePoint; //敵のHPを減らす
                        if (enemy.HitPoint <= 0) enemy.Dead(); //敵はHPが0以下で消滅
                    }
                    enemy.EndPlayerCollision();
                }
            }
        }
    }
}
```

ボタンを押すたびにプログラムが停止

enemy.HitPointの値を確認しようとするたびにプログラムを停止

双方向性，実時間性が高いプログラムのデバッグで共通する問題点

本研究の目的

◆本研究の目的

- プログラムを停止させないデバッグ手法を提案
- 双方向性，実時間性の高いプログラム一般に対し，汎用的に広く使える手法を目指す

提案するデバッグ手法

◆設計方針

- 一定間隔(例：0.1秒ごと)でプログラムの内部状態を可視化
  - プログラムの実行経路情報を実時間でハイライト表示
  - 開発者が監視対象として指定した変数の値

◆特徴

- 3段階の粒度で実行箇所を可視化する
  - 3段階：ソースコードファイル，メソッド，ステートメント

➔確認したい箇所を徐々に絞り込める

- 監視区間と条件を指定し，できる
  - 区間=開始ステートメント，終了ステートメントで指定

➔着目したい実行経路，変数値のみ観察できる

（例）for,foreachにおける特定要素，特定のスレッドが実行したとき

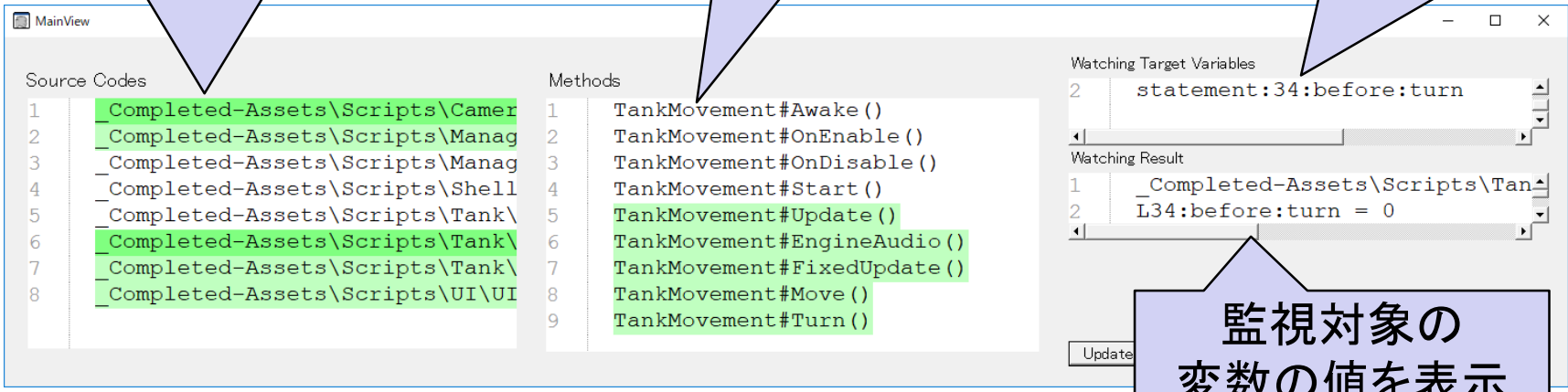
デバッガの画面構成

ソースコードファイル
ハイライト表示

メソッドハイライト表示

監視対象の変数を記述
監視区間と条件を記述

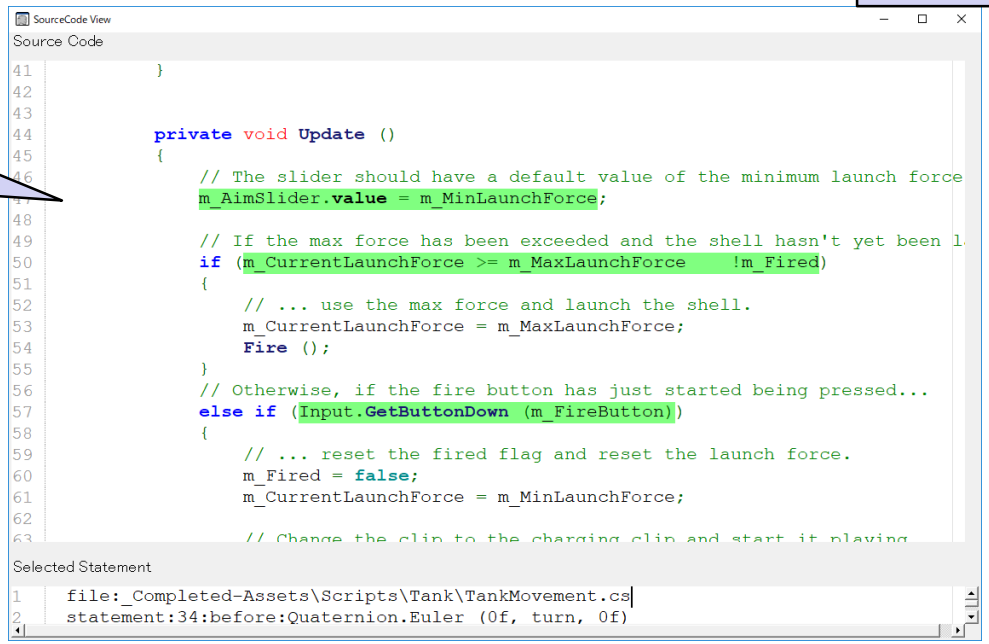
メイン
ビュー



監視対象の
変数の値を表示

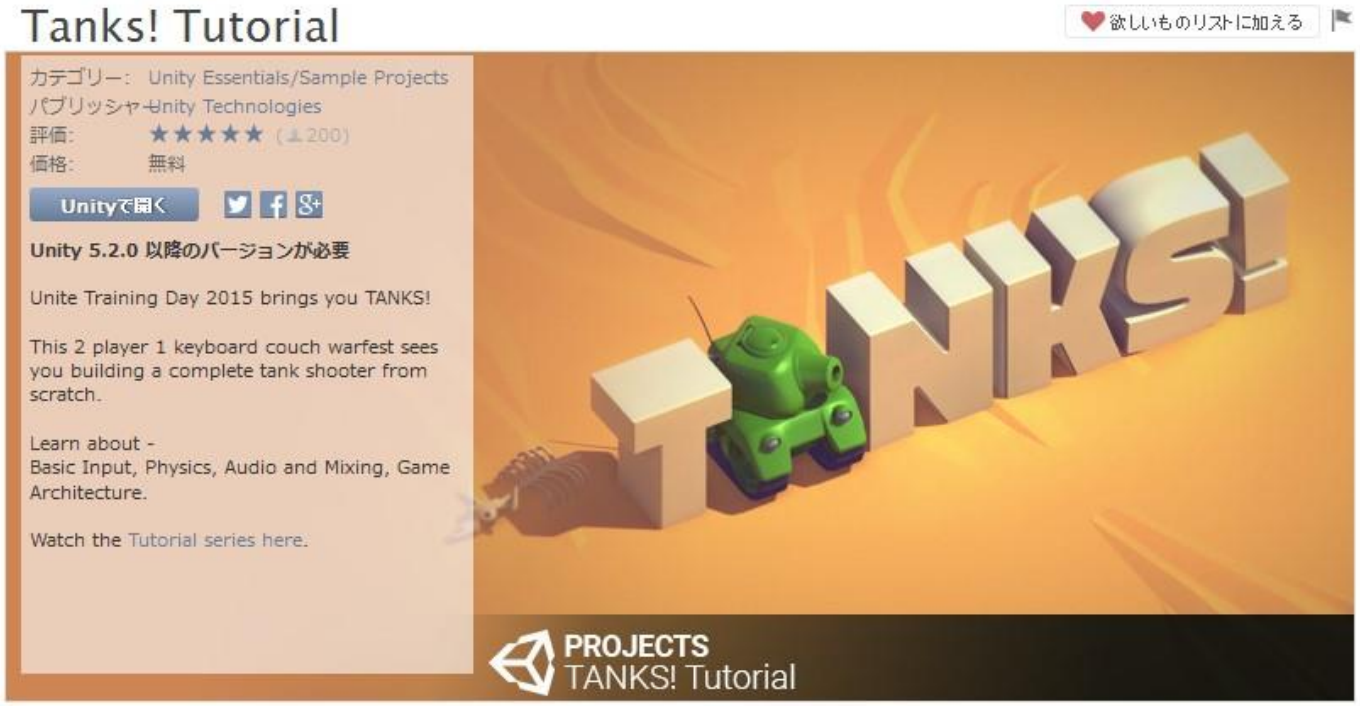
実行経路の
ハイライト表示

ソースコード
ビュー



提案デバッガのデモ

- ◆機能紹介
- ◆活用事例紹介



題材：Unity Tanks! (1.5KLほどのC#プログラム)
・戦車同士で撃ち合って戦うゲーム

提案手法の特徴(再掲)

◆特徴

- 3段階の粒度で実行箇所を可視化する
  - 3段階:ソースコードファイル，メソッド，ステートメント

➔確認したい箇所を徐々に絞り込める

- 監視区間と条件を指定し，できる
  - 区間=開始ステートメント，終了ステートメントで指定

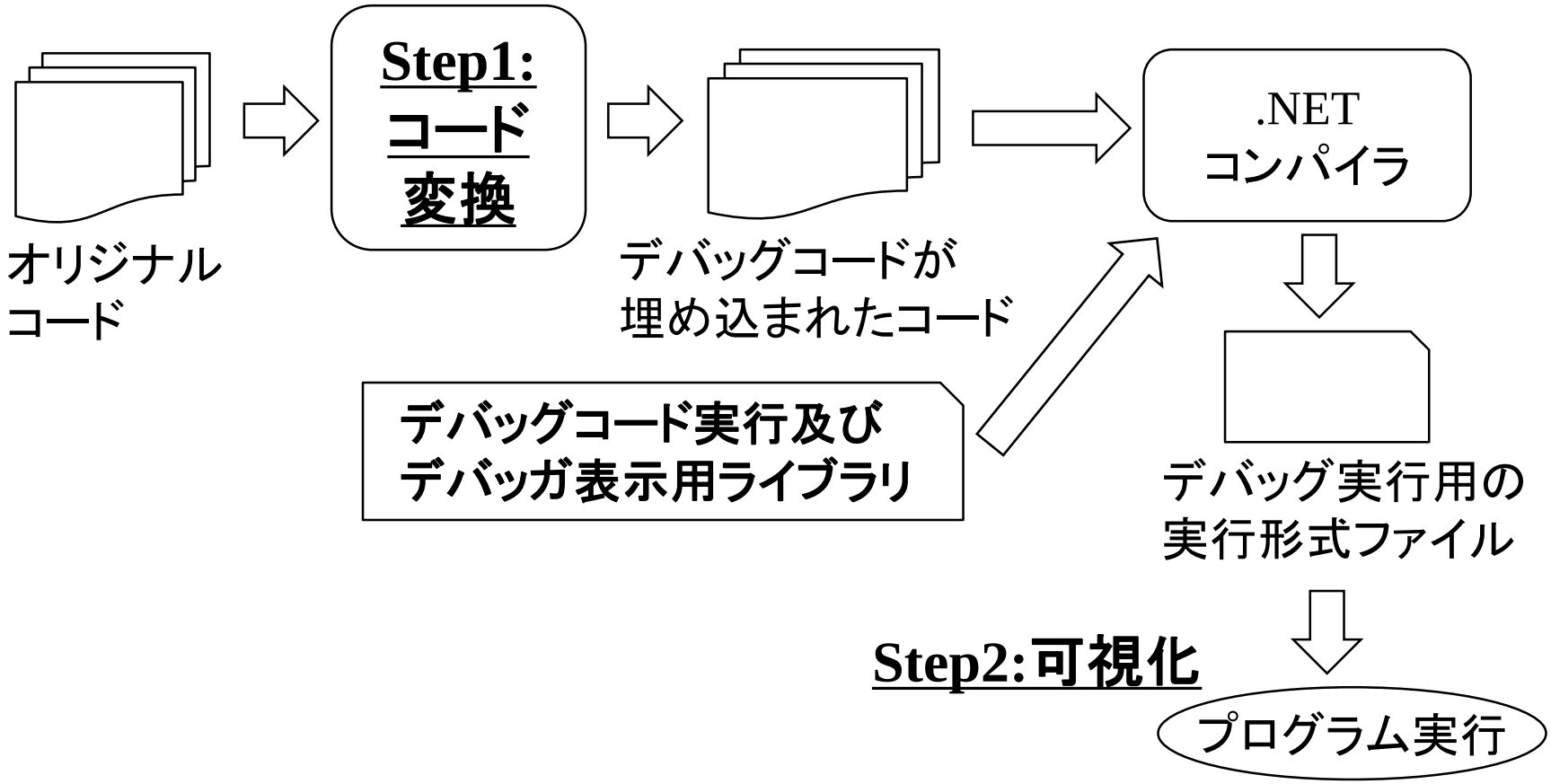
➔着目したい実行経路，変数値のみ観察できる

(例)for,foreachにおける特定要素，特定のスレッドが実行したとき

<u>➔双方向性，実時間性の高いプログラムを効率よくデバッグ可能</u>

C#における提案デバッガの実装方式

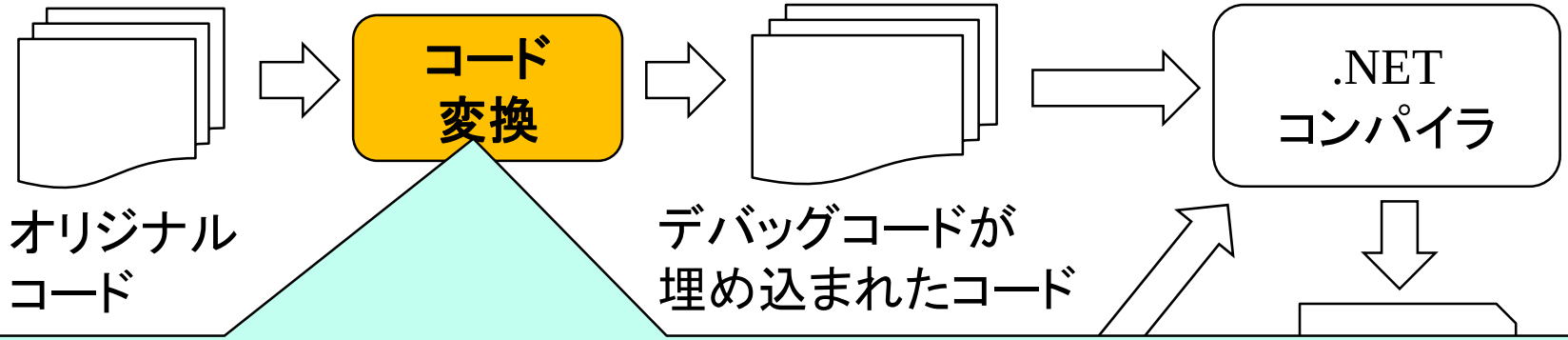
- **Step1：デバッグ対象コードをデバッグ用コードへ変換**
- **Step2：プログラム実行中にデバッグ情報を取得して可視化**



ユーザはデバッグ用ライブラリをリンクするだけで使用可

➔負担なく使える

Step1：デバッグ対象コードをデバッグ用コードへ変換



全てのトップレベルの
ステートメント/式を
ログ取得用メソッドで置換

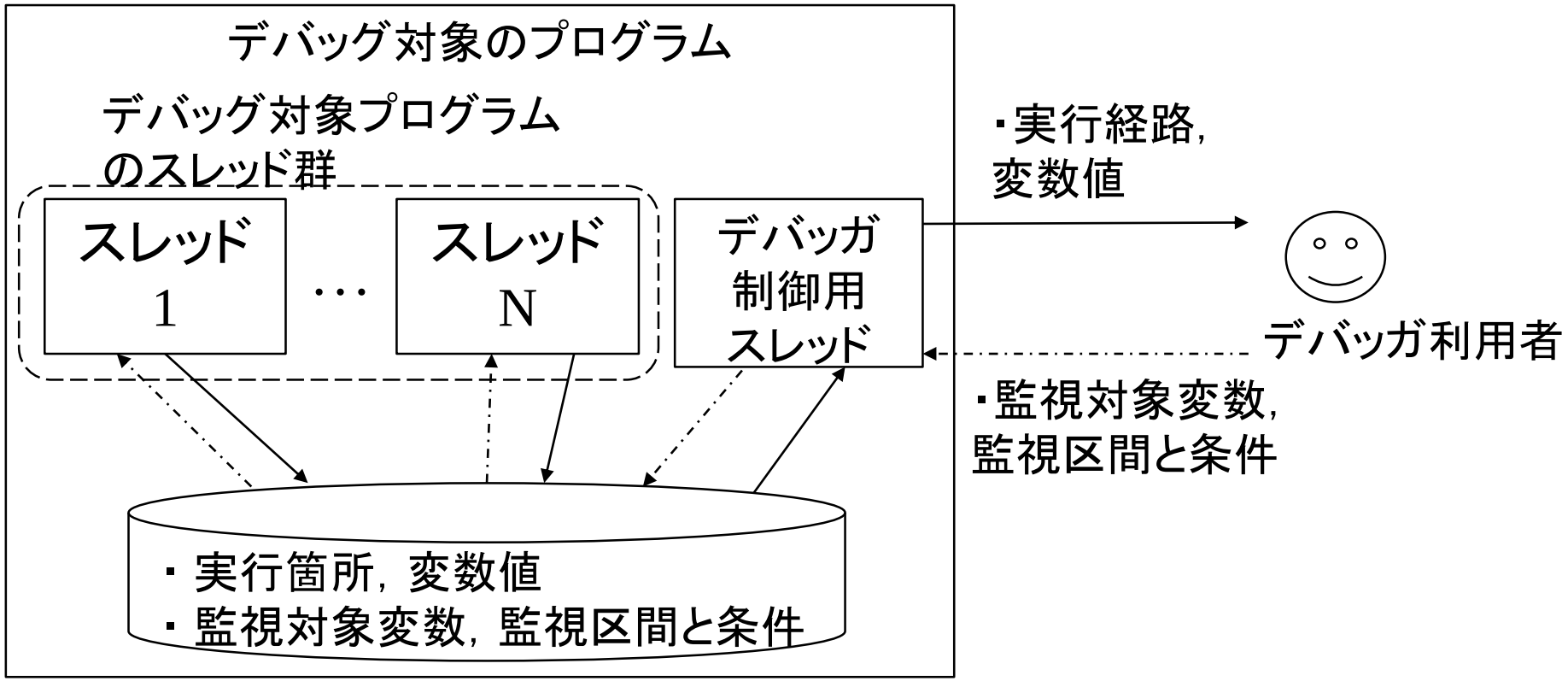
```
…
int damagePoint = player.OffensivePower;
if (input.AttackButton){
        if (input.DashButton)
            _damagePoint *= 5;
        foreach (var enemy in enemyList){
…
```

```
public T LogFunc<T>(
 Func<T> state, int stateId)
{
  デバッグ情報の取得();
  return state();
}
```

```
…
int damagePoint = _l.LogFunc(() => player.OffensivePower, 0);
if (_l.LogFunc(() => input.AttackButton, 1)){
        if (_l.LogFunc(() => input.DashButton, 2))
            _l.LogFunc(() => damagePoint *= 5, 3);
        foreach (var enemy in _l.LogFunc(() => enemyList, 4)){
…
```

Step2：プログラム実行中にデバッグ情報を取得して可視化

- デバッグ制御用スレッドは対象プログラムのスレッドの1つとして動作
- 対象プログラムのスレッドは実行経路，変数値をデバッグ用コードで記録

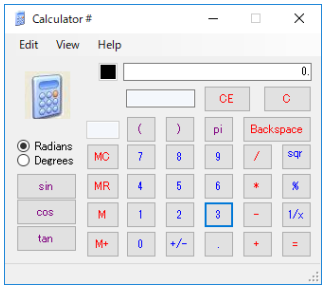


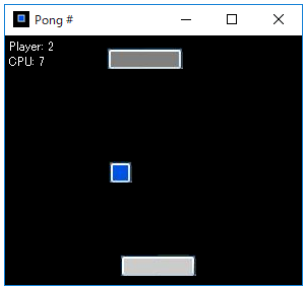
オーバーヘッド計測

◆ 計測対象

- GUIプログラム：電卓アプリ(2.5KL)
- ゲームプログラム：ピンポンゲーム(0.38KL)
- 数値計算：Fib(20)を計算

電卓アプリ ピンポンゲーム





◆ 計測結果

	実行回数	オリジナル (単位:ms)	デバッグ用(単位:ms)
GUI	100,000	7,360(100%)	10,344(**141%**)
ゲーム	1,000	1,632(100%)	1,848(**113%**)
数値計算	5,000	3,002(100%)	1,112,533(**3,749%**)

考察

- 数値計算は1ステートメントの計算量が少ないためオーバーヘッド大
  - （例）「n < 2」
- GUI，ゲームは1ステートメントの計算量が大きいためオーバーヘッド小
  - （例）「s.Substring(i, 1).Equals(Convert.ToString(j)) && i > lastPos」
- ゲームは描画ライブラリの計算量が多いため特にオーバーヘッド小

デバッグ手法の関連研究

- **ログベース**
  - ログを自動収集し可視化
    - （例）[Hermanら2018]：ログからデータフローと処理タイミングを可視化
  - 確認は実行後なので様々な入力で試行錯誤しにくい
  - 全情報のログはとれず，必要な情報が後から確認できない
- **プログラム実行再現**
  - プログラムへの入力を記録し，プログラム実行を再現
    - （例）C言語[Kojuら2005]，Java[Barrら2014]，JavaScript/Node.js[Barrら2016]
  - 確認は実行後なので様々な入力で試行錯誤しにくい
  - 完全な再現は技術的障壁が高く，制約により適用範囲が狭まることもある
    - [Barrら2014]ではマルチスレッド使用時の再現が保証できない
- **プログラムを停止させないデバッグ手法**
  - ゲームシステム記述に特化した言語における実装[丹野2008]
  - ドメイン特化言語を対象とする方式であり汎用性が低い

まとめ

- **プログラムを停止させないデバッグ手法を提案**
  - 3段階で実行箇所を可視化する
  - 複数回実行される箇所へ監視区間と条件を指定できる
  - マルチスレッド対応
- 提案手法をC#プログラムのデバッガとして実現
- 活用事例を紹介

- 今後
  - 実用プログラムを題材として提案手法の有効性を確認する