汎用ゲームエンジンに外付けするデバッグ環境の提案

茂原敦之† 水口充†

本研究では，ゲームデバッグの手法の一つとして外部プログラムからキャラクタを制御する方法を提案する．コントローラの状態を取得するクラスのラッパークラスを作成し，デバッグモードではコントローラの代わりに外部プログラムからの値を返すようにする．この方法を用いれば，既存の開発環境や開発途中のゲームプログラムに大幅に手を加えなくともデバッグ環境を外付けすることができる．また，必要とするデバッグ内容に応じて外部プログラムを変更，もしくは追加して作成すればよい．外部プログラムにはプレイログや AI など多様なデバッグ方法を利用できる．

1. はじめに

近年オープンワールドゲームなど，コンピュータゲームの規模がどんどん大きくなってきている．しかし，ゲームの規模が大きくなるにつれて，デバッグにかかる時間・コストが大きな問題となっている．ここでいうゲーム業界用語としてのデバッグとは，バグを直すことではなく，バグを検出すること(テストプレイ)を指す．バグはプログラムの不具合だけでなく，データの不備も含まれる．

従来のデバッグは，デバッグ作業を専門とする会社，もしくは開発したゲームメーカー自身が多くの人を集め，開発中のゲームソフトを実際にプレイしてもらい,不具合(バグ）を見つけるという手法である．しかし，人間によるデバッグでは多くのデバッガを集めて長時間テストしなければならない．

大手ゲームメーカーでは，自社開発でゲームデバッグ環境を作成し，デバッグ専用プログラムを用いてデバッグ作業を行なっている企業もある．しかしこのようなデバッグ環境を作成し，ゲームエンジンに組み込み，使用しているのは一部の大手ゲームメーカーであり，中小のゲームメーカーでは，デバッグ環境を作ることは困難であると考えられる．また，大手ゲームメーカーでのデバッグ環境を組み込むアプローチも内製のゲームエンジンで開発の初期段階から組み込む必要があり，特定のゲーム専用として作られてしまう．汎用のゲームエンジンで使えて，できるだけ開発の妨げにならないデバッグ環境が必要である．

本研究では，ゲームデバッグの手法の一つとして外部プログラムからキャラクタを制御する方法を提案する．コントローラの十字キーなどのキャラクタの移動や，ボタンなどのアクションを外部プログラムから制御する．そして，必要とするデバッグ内容に応じて外部プログラムの中身を変更,もしくは追加して作成すれる.この方法を用いれば，既存の開発環境や，開発途中のゲームプログラムに大きく手を加えなくともデバッグ環境を組み込む事ができる．また，プレイログや AI による多様なデバッグ手法にも利用可能となる．

2. アプローチ

2.1 要求されるデバッグ

ゲーム開発時に要求されるデバッグ項目について述べる．

ポリゴン抜け：

ゲーム内に設置された木，壁，建物などのオブジェクトをキャラクタが通り抜けてしまう現象．ポリゴン間の境界で，視覚的には繋がっているが物理演算的には隙間ができており，通れてはいけないはずが通れてしまう不具合が発生する。例えば，レースゲームでコース外を無限に走れてしまったり，オープンフィールドタイプのゲームで地面の中に侵入できてしまい，正常なゲーム進行ができなくなるという問題が発生する．

フレームレート低下：

単位時間あたりに処理させるフレーム数（静止画像数，コマ数）のことであり，ゲームではオブジェクトの密集度の高い場所にキャラクタが移動した際に，あまりの多さに処理が間に合わず，フレーム数が減り，処理落ちなどが発生してしまう現象．フレームレートが低下すると，例えばアクションゲームなどではタイミングが取りづらくなりプレイアビリティが低下してしまうし，多くのゲームでグラフィクスの品質の低下によりユーザ体験を失うことになる．

予期しないショートカット：

開発側が意図して作ったものではないショートカット．レースゲームで螺旋状のコースがあった時，コースを下に降っていくときにわざとコースアウトする事で，下に続いているコースにショートカットする．このようなショートカットは開発者が意図的に仕込む場合もあるが，意図しないショートカットが可能であることが発覚するとゲーム自体が破綻しかねない．

例として上記のものを上げたが，これらの項目以外にも要求されるデバッグは数多く存在し，ゲームの内容に応じて要求されるデバッグも変わる．統一的な手法で環境を整えられれば，多く存在するデバッグに応じて個別にデバッグ環境を作成していくという手間は減る．

† 京都産業大学大学院先端情報学部

2.2 従来のデバッグ手法

現状では人間の手によって，開発中のゲームソフトを実際にプレイし，不具合（バグ）を見つける手法が多くを占めている．あらゆる使われ方を想定し，仕事としてゲームを細かくチェックしていく．作業は長時間自由にプレイできるものもあれば，決められた部分を集中的にチェックするものもあり，中には長時間何もせずに放っておく，といった検査項目まで存在する[1]．

2.1 で挙げたようなデバッグに対しては,以下のような方法でデバッグを行っている．

ポリゴン抜けのデバッグ：木や壁などのオブジェクトに自ら接触し,抜けがないかを配置されている分だけ調べる．

フレームレート低下のデバッグ：フィールド内を走り回り，特に森などのオブジェクトが多くあるような場所，フレームレートが低下しそうであると考えられる場所にある程度の目星をつけ，その場所を色々な視点で移動する．

予期しないショートカット:コースの短縮ができないか，レースゲームでは，わざとコースアウトすることでショートカットができたり，RPG などのシナリオに沿って進めるゲームでは，シナリオ通りに進まずに一つ飛ばしで次のステージに無理やり進もうと試みる．

しかし人手によるデバッグは、何時間もモニターを見てやるので，目の疲れを感じて，時間が経つにつれ，集中力も低下していき，見落としが出てくる．似たような操作を何度も繰り返すことが多く，完璧に全てのバグを見つけ出すということを人間が行うのは労力を使う割にかなり難しい．

また，ゲームの規模が大きくなればなるほど，デバッグのために多くのデバッガ(テストプレイヤ)を長時間集めなければならなくなり，人件費などコストが大きくなってしまう．バグの出そうな場所を予測する事も必要になり，デバッガの質も問われる．

2.1 で挙げたようなデバッグに対して,人間がデバッグすることの問題点として共通していることがある．それは，似た操作を何度も繰り返さなければならないということである．

ポリゴン抜けのデバッグでは配置されているオブジェクト全てに衝突しに行く，フレームレート低下のデバッグではフレームレートは視覚に入るオブジェクトの数によっても変わってくるので，同じ場所を違う視点で何度も通る，予期しないショートカットでは，わざとコースを外れる，コースアウトするといった似た操作を何度も繰り返すため,全てを完璧に調べるということは人間には限界がある．

2.3 デバッグの自動化

近年、新たな手法としてデバッグの自動化が進められている。主にログデータを用いてのバグ発見や自動プレイによりデバッグを行う．プレイテストを行なってもらったときのログデータを集計し，自動プレイ用のパスを作成し，開発者が帰宅した後の PC やゲーム機を用いて，プレイテストを行う．ログデータからどの地点でキャラクタが倒されたのかなどのデータから，ステージによっての難易度チェックや，壁などに衝突判定がされていないかなどの漏れを確認するコリジョンチェックにも使用できる．

2.1 で挙げたようなデバッグに対しては,自動テストプレイを用いている．ポリゴン抜けでは，一定時間同じ方向に移動を続けてオブジェクトにぶつかる，半径数メートルの範囲内で動き回るといった動きをさせ，ポリゴン抜けのあった場所には印をつけ，ログに書き込んで画像にして確認する．ショートカットではコース外を走らせる，コースアウトさせるようにし，その後に正規ルートよりも速くないか，コースアウトしたはずが，まだ進めているなどをログから調べだす．フレームレートに関しては，自動プレイでキャラクタにマップ上をひたすら移動させた場所のフレームレートの変化をヒートマップに起こし，処理の重い場所を確認する[2][3].

2.4 提案する手法

今回私が提案する手法は，外部プログラムからキャラクタを制御する方法である．

この方法は，ゲームのコントローラの十字キーなどのキャラクタの移動や，ボタンなどのアクションを外部プログラムから制御する．そして，行いたいデバッグ内容に応じたキャラクタ制御プログラムとして作成する。外部からプログラムを追加するだけで良いので，ゲームエンジンに大きく手を加えることもなくなる．また，プレイログや AI など，多様なデバッグ方法を利用できる．デバッグプログラムを用いる事で人手を使うよりも，人件費やコストが削減できる．

3. 実装

3D のフィールドをキャラクタが自由に歩き回る様な環境を構築し，この手法の実装の可能性を検証した．

3.1 環境構築

この環境の構築のために，ゲームエンジンでは Unity，プログラミング言語では C#を使用した．

本研究の目的は,「既存のゲームエンジンにできるだけ手を加えずにデバッグ環境を提供すること」である．Unity はメインの開発ツールとして開発に利用しているという会社も数多く，現状最もシェアが大きいゲームエンジンと言われている．また，Unity は，無料でも使用することができ，アマチュアにも使えるので，本提案手法は広範囲に利用可能となる．

3.2 フィールド

簡易的なフィールドを Unity で自作した．Terrain という広

く平坦なオブジェクトを設置し，Terrain アセットから地面に近い画像を見つけ，それを Terrain オブジェクトに貼り付け，地面に見えるようにした．さらにフィールドには，オブジェクトとして木を設置しておく．

図 1 および図 2 は，このようにして制作したフィールドである．

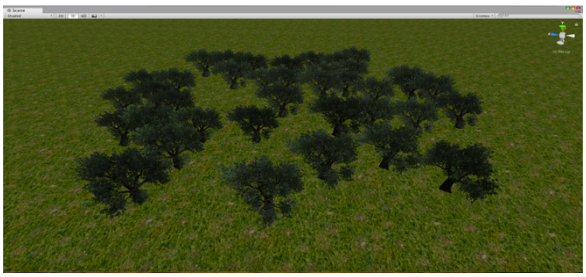

図 1　自作フィールド(上面)

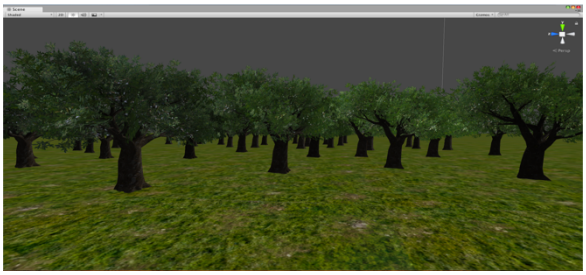

図 2 自作フィールド(側面)

3.3 キャラクタと移動

使用するキャラクタは，3D キャラクタモデルと移動スクリプトがセットになっている CharacterControllers アセットから，「3rd Person Controller」というキャラクタモデルを使用した．

キャラクタの移動も同様のアセットから「ThirdPersonController」というスクリプトを使用した．

「ThirdPersonController」ではキャラクタ移動として以下の変数を用いている．

```
float h = Input.GetAxisRaw(“Horizontal”);
float v = Input.GetAxisRaw(“Vertical");
```

変数 h と v はそれぞれ水平方向，垂直方向のキャラクタ移動量である．

Input クラスは，さまざまな入力を取得することができるクラスであり，その中に GetAxisRaw というジョイスティックやキーボードなど入力デバイスの移動量を，-1.0～1.0 の数値で取得する関数がある．使用しているキャラクタ移動スクリプトの「ThirdPersonController」はこの関数を用いて，キーボードでキャラクタを移動させている．”Horizontal”や”Vertical”は，正方向の入力を検知するボタンと，負方向の入力を検知するボタンを設定することができる．今回，”Horizontal”には，負方向の入力に左矢印キーが，正方向の入力に矢印右キーが，”Vertical”には，負方向の入力に下矢印キーが，正方向の入力に上矢印キーが設定されている．

図 3 はこのキャラクタの外見，図 4 は移動しているキャラクタである．

図 3 キャラクタとフィールド(側面)

図 4 キャラクタとフィールド(側面)

3.4 バーチャルコントローラ(外部プログラム)

キャラクタの移動を別クラスで制御する為に，ラッパークラス「MyInput」を作成した．

「MyInput」の GetAxisRaw 関数には以下のコードが書かれている．

```
public static float GetAxisRaw(string control){
        if (debugMode) {
                return 1.0f;
            } else {
                Input.GetAxisRaw(control);
        }
    }
```

debugMode が true の時はデバッグ用の制御，false の時は通常のコントローラに切り替える事ができる．

Input クラスの GetAxisRaw 関数をラップし，返り値を設定した．こうすることにより，入力デバイスではなく，「MyInput」スクリプトでキャラクタの移動を制御することができる．後は「ThirdPersonController」の「Input」を「MyInput」に書き換え，「MyInput」で移動の条件文を書き加えさえすれば，キャラクタを自由なタイミングで移動させたりすることができる．今回の場合は 1.0 を返しているので，”Horizontal”では，正方向の入力に設定されている矢印右キーが，”Vertical”では，正方向の入力に設定されている上矢印キーが押された時と同じ移動をする．

当初は，「ThirdPersonController」を継承し，GetAxisRaw 関

数の値を継承した別スクリプトで変えることで入力デバイスなしにキャラクタを動かせるのではないかと考えいたが，Input クラスには sealed 修飾子というクラス修飾子が付けられており，継承することができなかったため，ラッパークラスを用いることにした．

以上のプログラム構成を図示すると次のようになる．

図 5 は，標準の Unity の処理の流れである．

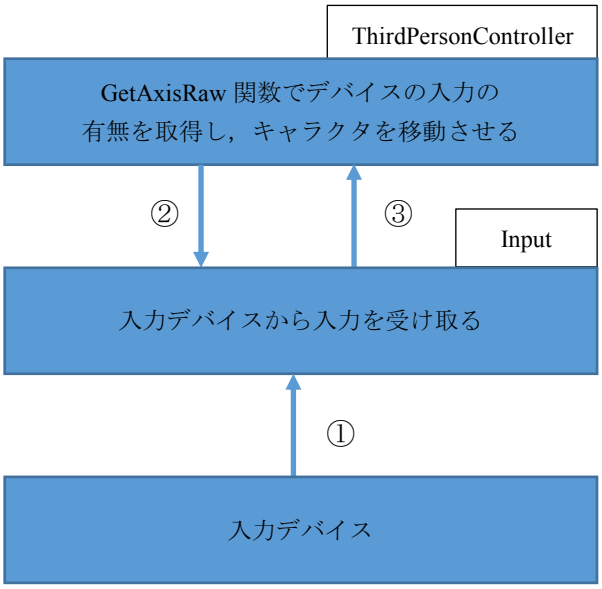

図 5 入力デバイスによるキャラクタ操作

以下の処理の流れで入力デバイスの入力でキャラクタを動かす．

①入力デバイスの入力を Input クラスが取得

②Input クラスに対し，値を要求する．

③入力デバイスから取得した値を，ThirdPersonController クラスの GetAxisRaw 関数に渡す

非デバッグモード

①入力デバイスの入力を Input クラスが取得

②MyInput クラスに対し，値を要求する．

③入力デバイスから取得した値を，ThirdPersonController クラスの GetAxisRaw 関数に渡す．

非デバッグモード時は，図 5 と同様のステップを踏む．入力デバイスからの値を ThirdPersonController クラスの GetAxisRaw 関数に渡す事で通常通り，入力デバイスからの操作でキャラクタを移動する．

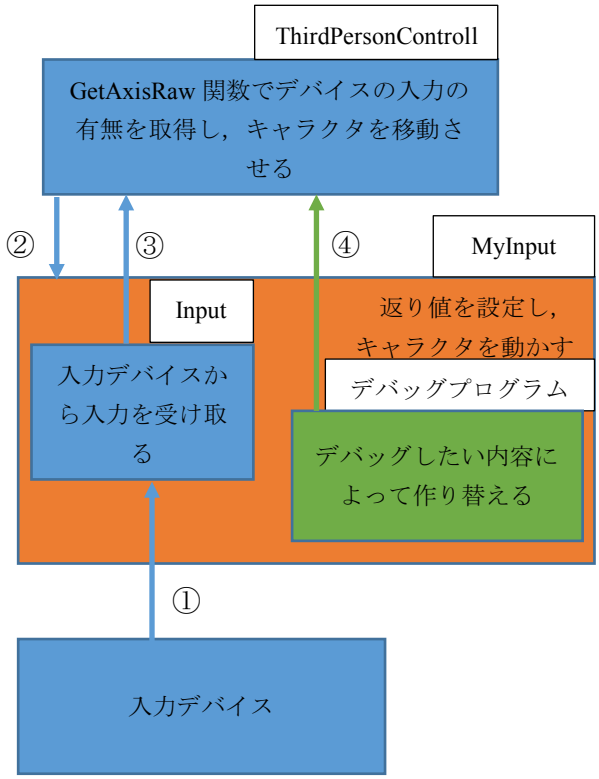

図 6 バーチャルコントローラによるキャラクタ操作

デバッグモード

②MyInput クラスに対し，値を要求する．

④MyInput クラスで設定した値を，ThirdPersonController クラスの GetAxisRaw 関数に渡す．

デバッグモード時は，②→④というステップを踏む．MyInput クラスで設定した返り値を ThirdPersonController クラスの GetAxisRaw 関数に渡す事で，MyInput クラスでキャラクタを移動させる．デバッグプログラムと連携し，デバッグさせたい内容によって返り値を変更していく．

このように，MyInput クラスは非デバッグモードに設定すれば Input クラスへの呼び出しをそのまま受け渡すので，デバッグ用のコードが開発中のプログラムの不具合の原因になる事を防げる．

3.5 デバッグプログラム

挙動の検証用にデバッグプログラムを作成した．このプログラムは，フィールド上にあるオブジェクト(木)の位置を調べ，その位置まで移動し，その周りをキャラクタが体をこすりながら何週も走るといったものである．

このプログラムはポリゴン抜けを想定し，作成した．

図 8 自作デバッグプログラムの挙動

図 8 はフィールドを上から見たものである．中央にキャラクタが立っており，デバッグをスタートさせる気に向かって走り出す．

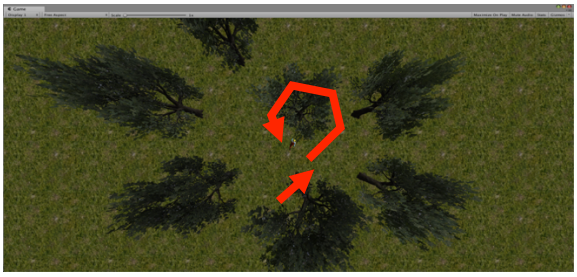

図 9 自作デバッグプログラムの挙動 2

図 10 自作デバッグプログラムの挙動 3

図 9 では，木に向かって走り出しており，木に着くと周りを体をこすりつけながら周回する．その後，図 10 のように別の木に向かって走り出し，その木に着くとその周りを周回．それを繰り返すようなプログラムになっている．

4. 考察

実装したキャラクタ制御プログラムにより，キャラクタの移動を制御することに成功した．また，作成したデバッグプログラムと組み合わせることで，フィールドに設置した木の場所を自動で特定し，全て回りきる挙動が実現できた．

検証用に作成したデバッグプログラムは，設置した木がフィールドのどこの座標に植えられているのかという位置情報を全て取得し，設置されている木に総当たりで調べるといったものになっている．オブジェクトの位置情報がフィールドから取得できるということから，当たり抜け以外にも木の位置情報から木の密集度を測り，密集した場所を重点的に移動し，その場のフレームレートの変化度合いを調べるといったデバッグプログラムも作れるのではないかと考える．

本提案手法により外部プログラムでキャラクタを制御する事が出来たので，この手法を用いてできる他のデバッグや，デバッグ以外の用途を模索していきたい．

5. まとめ

本研究では，既存のゲームエンジンと開発中のゲームプログラムにできるだけ手を加えずにデバッグ環境を提供する手法を提案した．この手法により，外部から追加したキャラクタ制御プログラムに書きこみ，デバッグしたい事柄によって内容を変更，もしくは追加することで，容易にデバッグ環境を作成できる．さらに，この方法とキャラクタ移動を指示するプログラムを用いることで，ポリゴン抜け，フレームレート低下といった人間がすれば見落とすことが多いであろう似たような操作を何度も繰り返すことを要求されるデバッグを自動化することも可能になる．

謝辞

本研究を進めるにあたって，きっかけを与えていただいた株式会社トーセ開発本部の大下岳志様，森本健様，岩倉高征様，時田康孝様，城尚嗣様，心より感謝申し上げます．

参考文献

［1］ ボムス Q&A 相談室
https://www.baitoru.com/contents/bm_faq15/20.html
[2] GameWatch【CEDEC2017】「龍が如く」が毎年発売される理由の一端はここにある？
https://game.watch.impress.co.jp/docs/news/1078467.html
[3] 阪上直樹，無料で始める!「龍が如く」を面白くするための高速デバッグログ分析と自動化
http://jasst.jp/symposium/jasst18tokyo/report.html#research
［4］ 三宅陽一郎，ディジタルゲームにおける人工知能技術の応用の現在．