テトリスのためのInpaintingを用いたルールベースなデバッグAI

髙橋　秀太朗† 服部　峻†† 砂山　渡††

† 滋賀県立大学 工学部 電子システム工学科　〒522–8533 滋賀県彦根市八坂町 2500
†† 滋賀県立大学 先端工学研究院　〒522–8533 滋賀県彦根市八坂町 2500
E-mail: †on23stakahashi@ec.usp.ac.jp, ††{hattori.s,sunayama.w}@e.usp.ac.jp

あらまし　ゲーム開発において，デバッグに掛かる時間・コストが増大しており，これらを下げるべく人間のデバッガーの代わりにバグを判別可能なデバッグ AI を構築するためには，プレイ動画からのバグ発見や，効率的なバグ発見のためのプレイ操作系列の自動生成など課題は多い．そこで本稿では，テトリスゲームを題材に，人間プレイヤー操作によるバグ発生を含むプレイ動画を画像認識して，OpenCV のテンプレートマッチング，Inpainting などの機能を用いて構成されたルールベースでバグを発見するデバッグ AI を試作し，その性能を検証する．

キーワード　画像認識，ゲーム情報学，デバッグ AI，ルールベース，Inpainting，テンプレートマッチング

1 はじめに

近年，ゲーム開発の規模が年々拡大してきており，2020 年には日本国内のゲームコンテンツ市場が 2 兆円を上回り，過去最高の規模となっている．2021 年はほぼ横ばいの状態であり，高い市場規模を維持している [1]．それに伴って，デバッグに掛かる時間・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を指す [2]．

従来のデバッグ方法の基本的な手順として，ゲームに長けた人間のデバッガーやテスターがプレイヤーの視点に立ってプレイを行い，その中でゲームに含まれていた不具合などのバグを発見するというものである．

ゲーム開発のデバッグに掛かる時間，コストを下げるべく，人間のテスターの代わりにバグを判別可能なデバッグ AI を構築するには，プレイ動画からのバグ発見や，効率的なバグ発見のためのプレイ操作系列の自動生成など課題は多い．これらの問題の解決策として，ルールベースや機械学習が応用できるのではないかと考えた．

そこで本研究では，テトリスゲームを題材にして，人間プレイヤー操作によるバグ発生を含むプレイ動画を画像認識して，ルールベースでバグ発見するデバッグ AI を試作して，その性能を検証してきた [3]．

発見可能なバグの種類を増やし，ルールベースなデバッグ AI の汎用性を高めるためには，ブロックや枠など，プレイ動画中のオブジェクトを精度良く認識する必要がある．そこで本稿では，画像認識の 1 種のテンプレートマッチングに加え，オブジェクトの重なり問題を解決するため，画像修復の Inpainting も組み合わせる．

以降，2 章では本研究の関連研究を紹介する．次に 3 章では，求められるデバッグ AI の要件分析を行う．4 章では，本研究の提案手法を示す．5 章では，評価実験を行って，提案手法の性能を評価する．最後に 6 章でまとめと今後の課題を述べる．

2 関連研究

本章では，関連研究について述べる．

2.1 デバッグ環境に関する研究

汎用ゲームエンジンに外付けするデバッグ環境の提案を行う研究 [2] がある．この研究では，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ゲーム内のキャラクタを制御する方法を提案している．この方法を用いることで，既存の開発環境や開発途上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可能となる．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ることができる．

本研究との違いとして，関連研究はゲーム内のキャラクタの制御などのデバッグ環境を AI に任せるが，最終的にバグかどうかを判断するのは人間であるのに対して，本研究の目標は最終的にプレイ操作系列を自動生成し，さらにプレイ動画を自動的にデバッグを行うものとしており，最終的にバグかどうか判断することを AI に任せる点が挙げられる．

2.2 ゲームの裏技や不具合を発見する研究

アクションゲームの裏技を強化学習によって発見する研究 [4] がある．この研究では，スーパーマリオブラザーズの裏技の一つである「無限 1UP」を，進化戦略と多層パーセプトロンを用いた深層強化学習によって発見する AI を作成している．

また，ゲームシナリオの不具合を発見するツールを提案する研究 [5,6] がある．この研究では，モデル検査ツール SPIN を用いて，ゲームシナリオをそのまま検査できるツールを提案して，シナリオの不具合を発見している．

本研究の類似点として，ゲームを題材にしている点，不具合や裏技などバグと言えるようなものを扱っている点が挙げられる．本研究では，テトリスを題材として，OpenCV を用いてゲーム画面を認識させ，ルールベースでバグ発見するデバッグ AI を作成することを目的とする．

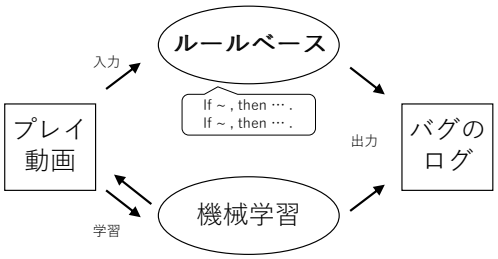

図 1　ルールベースと機械学習の違い

3　デバッグ AI の検討

本章では，本稿の対象ゲームであるテトリスだけでなく，汎用的なデバッグ AI を構築するための手法を検討する．

そもそも，ゲームにはなぜバグが混入してしまうのか，以下のような理由が挙げられる．

- 要求仕様書がそもそも曖昧で良くない
- システム開発の人員や工数が不十分である
- システムのテストが不十分である
- スマホゲームやシステム開発のサイクルが早い
- アップデートに甘んじて，開発をないがしろにする

3.1　手法の検討

本研究の最終的な目標は，汎用的なデバッグ AI の構築である．そのためには，効率的なバグ発見のためのプレイ操作系列の自動生成なども必要であるが，本稿では人間プレイヤー操作によるバグ発生を含むプレイ動画からのバグ発見のみを検討すると，図 1 のように，

- ルールベース
- 機械学習

を用いた手法が考えられる．

ルールベースを用いた手法とは，テトリスを例にすると，3.2 節で述べるような要求仕様書から予知できるバグや，人間のデバッガーやテスターなどによって既知となったバグの一つ一つをあらかじめルールとして記述しておくことで，バグが発生した際にどのバグが発生したのか検出することができる．

一方で，機械学習を用いた手法の例として，バグの有無をラベル付けしたプレイ動画を何度も教師付き学習させることによって，幅広いバグに対応させる手法が考えられる．但し，本稿では扱っておらず，今後取り組んで行く予定である．

3.2　バグの分類

本研究を行う上で，バグを以下の 3 つに分類する．

- 仕様書が明確で，仕様書から予知できるバグ
- 仕様書が曖昧で，予知できなかったが，人間のデバッガーやテスターなどによって既知となったバグ
- 存在するかもしれない未知のバグ

例えば，本稿の実装で使用した Java 版 Tetris [7] の場合，要件 (requirements) として，以下のように記述されている．

『Task Create a playable Tetris game.

Requirements

1. a left / right key
2. a hard drop key (the current piece will be dropped and locked at once)
3. a rotation key
4. a preview piece
5. full set of 7 kinds of shapes (ITOSZJL)』

これらの要件から，以下のようなバグ（ルール）が予知できる．

1a. if 左キーを押したにもかかわらず，ブロックが（左壁は除き）左に移動しなければ，then バグが発生している．

1b. if 右キーを押したにもかかわらず，ブロックが（右壁は除き）右に移動しなければ，then バグが発生している．

2. if 下キーを押したにもかかわらず，ブロックが一番下まで移動しなければ，then バグが発生している．
3. if 上キーを押したにもかかわらず，ブロックの向きが回転しなければ，then バグが発生している．
4. if 次に落ちて来るはずのブロックとは異なるブロックが落ちて来たら，then バグが発生している．
5. if 7 種類の形のブロックが落ちてこなければ，then バグが発生している．

しかしながら，要件に記述されていないため，

6. if ブロックが 1 ライン（横に 10 個）揃ったにもかかわらず，そのラインが消えなければ，then バグが発生している．

といった既知のバグは要件からは予知できず，別途バグルールを検討する必要がある．また，未知のバグが存在するかもしれないが，ルールベースでは対応できない．

3.3　プレイ動画分析

ルールベースを用いた手法において，発見したいバグに依って，プレイ動画の分析方法が異なる．

- 1 枚の静止画ごとに分析すれば良いバグ

「常にあるべき表示が無い」バグや，「ラインが揃ったのに消えない」バグなどは，プレイ動画の 1 枚の静止画であるフレーム毎にルールベースでバグ判別することで解決できると考える．

- 局所的な数枚の静止画を分析する必要があるバグ

「左/右キーを押したにもかかわらず，その通りに動かない」バグの場合，1 枚の静止画だけではバグ判別できず，少なくとも連続する 2 枚の静止画の差分が必要になる．また，プレイ動画だけでなく，その瞬間のプレイ操作（左/右ボタン押下）との照合も必要になる．

- 数ゲーム分の動画が必要となるかもしれないバグ

「7 種類の形のブロックが全て落ちてこない」バグの場合，少なくとも 7 個のブロックが落ちてくるまでは，バグが混入しているか分からないため，局所的な枚数の連続する静止画でも不十分である．実際には，大量のプレイ動画が必要となるかも

図 2　テトリスのブロックが横に 10 個並んだテンプレート画像

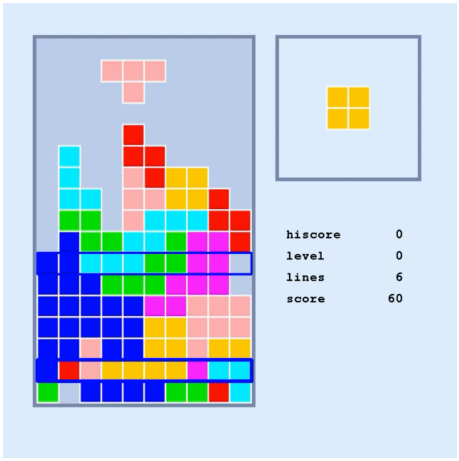

図 3　プレイ動画中の静止画における図 2 のテンプレート画像のマッチング（赤枠）とバグ発見（青枠）

図 4　テトリスのブロック 1 個ずつのテンプレート画像

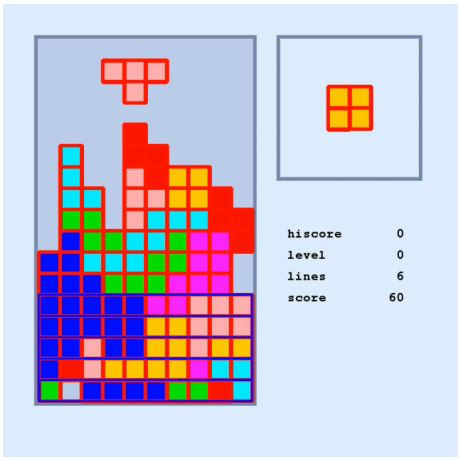

図 5　プレイ動画中の静止画における図 4 のテンプレート画像のマッチング（赤枠）とバグ発見（青枠）

しれない．例えば，100 個のブロック落下でも 7 種類のブロックが揃わない確率は，$1.41 \cdot 10^{-6}$ であり，閾値以下で十分に起こり難いとしてバグ発見する方法などを検討している．

4　提 案 手 法

本章では，前章の検討を踏まえ，テトリス用のデバッグ AI をルールベースなゲーム画面認識によって試作する．

本稿で試作するデバッグ AI では，プレイ操作との照合は行えないため，キーの押下に関連するバグには対応できず，プレイ動画の画像認識だけで判別可能なバグに限定され，

- 「ラインが揃ったのに消えない」バグ
  (Line Bugs と呼ぶ)
- 「ブロックが枠外に移動できてしまう」バグ
  (Can Move Bugs と呼ぶ)

をルールベースで検出することを試みる．さらに，図 1 のバグのログとして，入力されたプレイ動画のフレーム毎，これらのバグが発生している箇所の総数が出力される．ここで，枠とは，図 3 のプレイ画面の左の枠，10 · 17 のブロックが移動できる背景が灰色の領域のことである．

3.3 節の検討から，「ラインが揃ったのに消えない」バグと「ブロックが枠外に移動できてしまう」バグはそれぞれプレイ動画中の 1 枚の静止画であるフレーム毎に「ラインが揃っている」か否か，「ブロックが枠外に存在している」か否かを判別すれば良く，画像認識技術の一つであるパターンマッチングが活用できる．本稿では，Python 版 OpenCV のテンプレートマッチング関数 cv2.matchTemplate() を用いる．

4.1　「ラインが揃ったのに消えない」バグ

まずは直感的に図 2 のようなブロックが横に 10 個揃ったテンプレート画像を使用する．テンプレートマッチングの直前にグレースケールに変換される．プレイ動画中，テンプレートマッチングした Frame（動画時刻）に対して，図 3 のようにそのままバグが発見されたと判別する．

ここで，落下して来る 4 個のブロックの形（と色）には ITOSZJL と 7 種類あり，ブロックが横に 10 個揃ったパターンにもカラーバリエーションが存在するため，その全ての組み合わせをテンプレート画像（バグルール）とする手法も有り得る．しかし，バグルールが膨大になり，網羅的に用意するのも網羅的にマッチング処理するのも高負荷であるため，本稿ではテンプレート画像との類似度 sim の閾値の制御で解決を試みる．閾値を高く設定し過ぎると，図 2 のテンプレートと同じ箇所にしか正しくマッチングせず，一方で閾値を低く設定し過ぎると，図 2 以外のカラーバリエーションのブロックが横に 10 個揃ったパターンにもマッチングするだけでなく，全く関係の無い箇所にまでマッチングしてしまい，ノイズになる危険性があると予想される．図 3 は，図 2 のテンプレート画像を使用し，テンプレート画像との類似度の閾値が 0.90 の場合であるが，下から 7 列目においてバグを誤検知し，一方で，下から 3 列目から 5 列目において 3 ヶ所の「ラインが揃ったのに消えない」バグを見逃している．尚，図中のテンプレートマッチングによる赤枠とバグ発見の青枠は常に同じ場所であるため，重なっている．

ブロックが横に 10 個揃ったテンプレート画像はより直感的であり，もしも 1 種類用意するだけでも網羅的にバグ検出できるとなれば，そのままバグルールとして使い勝手が良かった．しかし残念ながら，図 3 のように，バグを見逃す偽陰性だけでなく，誤検知（偽陽性）まであり，事前実験でも精度が期待できなかったため，次の手法も導入する．

図 2 のようにブロックが横に 10 個揃ったテンプレート画像ではなく，図 4 のようにブロック 1 個ずつをテンプレート画像（7 種類）とする手法も有り得る．テンプレートマッチング自体はより高精度になると期待できるが，テンプレートマッチングされた箇所（赤色の正方形）が横に 10 個並んでいるか否か，追

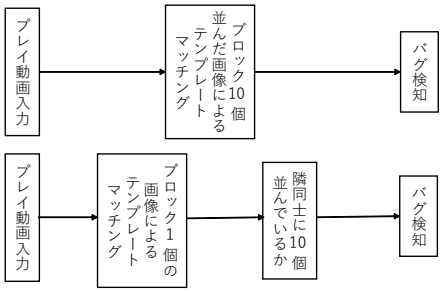

図 6　「ラインが揃ったのに消えない」バグの検知手法 2 種類のフローチャート図

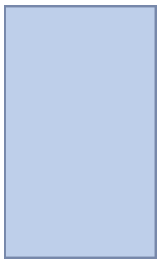

図 7　テトリスの枠のテンプレート画像

図 8　Inpainting 後における図 7 のテンプレート画像のマッチング（ピンク枠）

加的な処理が必要になり，煩雑である．図 5 は，図 4 の青色ブロックのテンプレート画像を使用し，テンプレート画像との類似度の閾値が 0.85 の場合であるが，図 3 と比べてバグの発見度合は高まったが，最下部にてバグの誤検知が発生している．

図 6 に処理の流れを示す．フローチャート図の上側が図 2 のテンプレート画像を用いた手法，下側が図 4 のテンプレート画像を用いた手法である．

4.2 「ブロックが枠外に移動できてしまう」バグ

4.1 節と同様に，まずは図 7 のような枠のテンプレート画像を使用する．ブロックが枠外に存在するかどうか判別するためには，プレイ動画での枠を認識させることは重要である．しかし，テトリスの仕様上，枠内にブロックが多く存在するため，プレイ動画ではテンプレート画像とかけ離れた状態が続き，閾値をかなり下げても枠を検出しない．

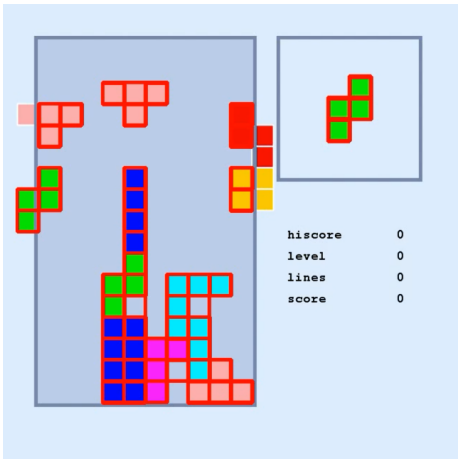

図 9　枠外バグ発生中の静止画における図 4 のテンプレート画像のマッチング（赤枠）

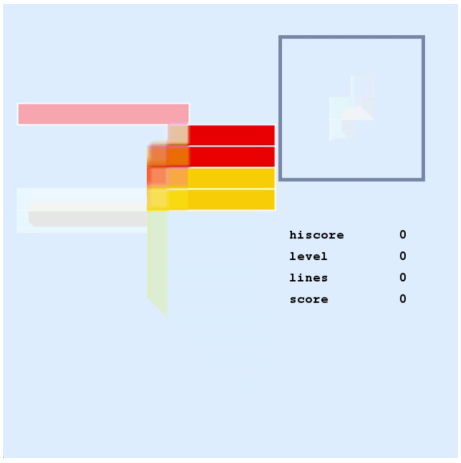

図 10　枠を Inpainting したプレイ画面

そこで，画像を修復する Inpainting という技術を導入する．本稿では，Python 版 OpenCV の cv2.inpaint() を使用する．これにより枠内のブロックを枠内の背景の色と同化させることが可能となり，元の枠のテンプレート画像を認識できるのではないかと考えた．

図 8 は，図 3,5 と同じタイミングで検知されたブロックを全てマスクした上で，それらの中を Inpainting したものである．画面下部が OpenCV の Inpainting の仕様上，枠の淵自体の色が影響して黒くなっており，閾値を 0.50 程度まで下げる必要があるが，枠を検知している．

図 9 は実際に枠外にはみ出たプレイ動画に図 4 のブロック 1 個で閾値 0.85 でテンプレートマッチングしたときの様子である．左の 2 つの枠外ブロックは検出しているが，使用したテトリスの仕様上，ブロックの上に枠が来るため，基本的にブロックとして認識できない．そこで，図 10 のように，テンプレートマッチングにより検知した枠でさらに Inpainting を行うことで，枠外にはみ出るブロックをより元のブロックに近づけさせ，ブロックとして認識できるようになるのではないかと考えた．尚，図 9 の左の枠外の Z ブロック（緑）の 2 個に相当する帯のようなものが図 10 には現れていないのは，枠を Inpainting

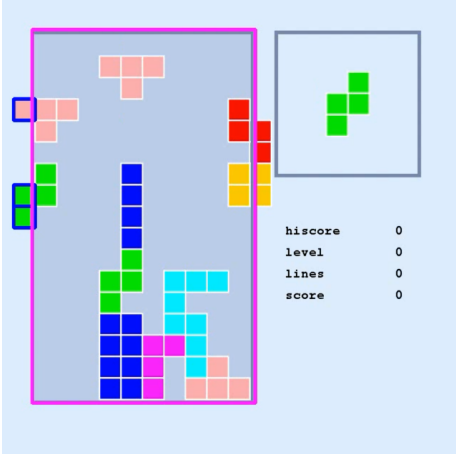

図 11　プレイ動画中の静止画における図 7 のテンプレート画像のマッチング（ピンク枠）とバグ発見（青枠）

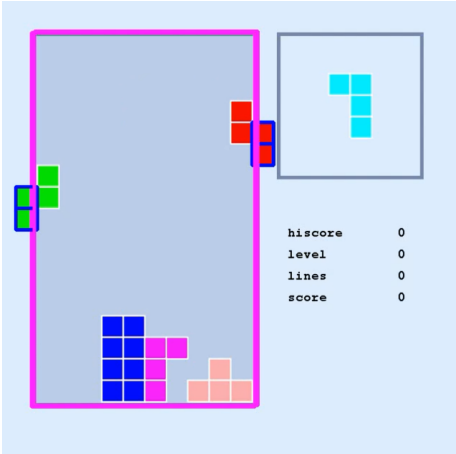

図 12　移動中のブロックが枠の上に存在している様子

する前に，左の枠外の Z ブロック（緑）が 1 つのブロックとしてテンプレートマッチングにより検知され，枠内ブロックを Inpainting する際に一緒に Inpainting されたためであると考えられる．

図 11 は枠外バグを検知している様子であり，図 4 の青色ブロックのテンプレート画像と図 7 の枠のテンプレート画像を使用し，ブロックのテンプレート画像との類似度の閾値が 0.85，枠のテンプレート画像との類似度の閾値が 0.50 の場合となっている．図 10 のような工夫を駆使しても，右側の枠外ブロックを検知することができず，左側の枠外ブロックのみ検知できている．尚，図 12 のように，移動中のブロックは枠の上に存在する仕様のため，右側のブロックも検知できる．

最後に，処理の流れをまとめると，プレイ動画のプレーム毎に以下を行う．また，図 13 に処理の流れを示す．

Step 1. ブロック 1 個のテンプレート画像を用いてテンプレートマッチングを行う．

Step 2. 枠内が背景と同じ灰色で塗りつぶされることを期待して，Step 1 で検出されたブロックの箇所をマスクして，Inpainting で画像修復する．

Step 3. Step 2 の結果を元に枠のテンプレート画像を用いてテンプレートマッチングを行い，プレイ画面の枠を検出する．

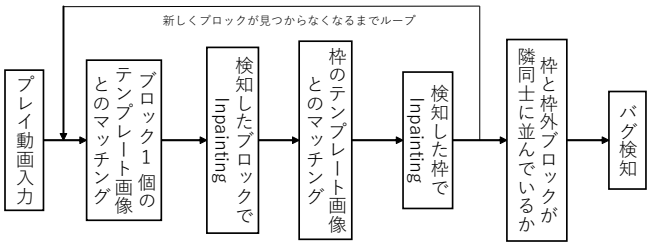

図 13　「ブロックが枠外に移動できてしまう」バグのフローチャート図

Step 4. Step 1 で検知できなかったブロックを検知するために，Step 3 で検出した枠をマスクして，Inpainting を行い，再度ブロック 1 個のテンプレート画像を用いてテンプレートマッチングを行う．

Step 5. Step 1 〜 4 を新しくブロックが見つからなくなるまで繰り返す．

Step 6. 検出した枠外のブロックと Step 3 で検出した枠が隣り合わせで並んでいるとき，「ブロックが枠外に移動できてしまう」バグが発生したと判別する．

5　評価実験

本章では，前章で提案した「ラインが揃ったのに消えない」バグと「ブロックが枠外に移動できてしまう」バグをそれぞれ発見できるテトリス用のデバッグ AI のプロトタイプの性能をそれぞれ検証し，さらにまとめて発見できるデバッグ AI のプロトタイプの性能も検証する．評価実験の手順は以下の通りである．

Step 1. Java 版 Tetris [7] を改造し，「ラインが揃ったのに消えない」バグ，「ブロックが枠外に移動できてしまう」バグを確率 0.30 で混入させ，プレイ動画を作成する．その際，バグ発生の動画時刻の正解ログも出力する．

Step 2. 提案手法を用いて，プレイ動画からバグを発見する

Step 3. 正解ログとバグ発見ログを比較する

5.1　「ラインが揃ったのに消えない」デバッグ

Frame#339，Frame#746，Frame#757 それぞれ 2 つずつ，「ラインが揃ったのに消えない」バグが発生し，それ以降は残り続けるプレイ動画に対して，2 種類の提案手法を適用して，ルールベースなゲーム画面認識によるデバッグ AI のプロトタイプの性能を検証すると，テンプレート画像との類似度の閾値に依って，図 14 と図 15 が得られた．

まず，図 14 から，ブロックが横に 10 個揃ったテンプレート画像（図 2）に基づくルールベースなデバッグ AI では，誤検知がかなり多く発生してしまっている．また，テンプレート画像との類似度の閾値を 0.80 よりも大きくしてしまうと，全く無反応になってしまう．逆に，閾値を小さくし過ぎると，誤検知

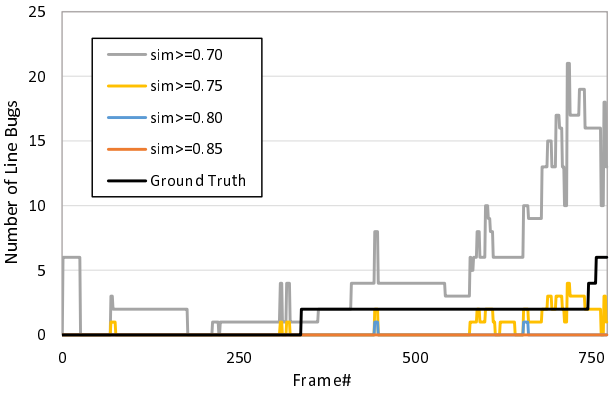

図 14　ブロックが横に 10 個揃ったテンプレート画像（図 2）に基づくルールベースなデバッグ AI のバグ検出数

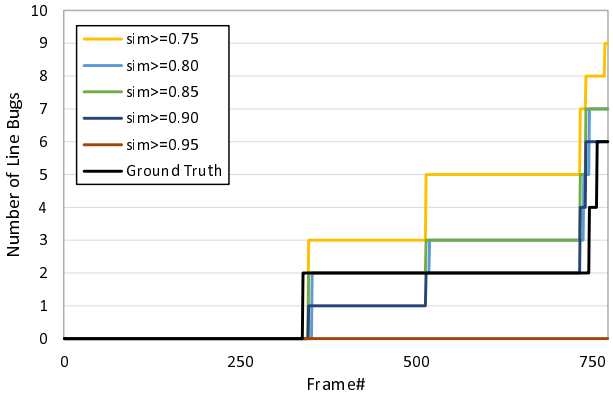

図 15　ブロック 1 個ずつのテンプレート画像（図 4）に基づくルールベースなデバッグ AI のバグ検出数

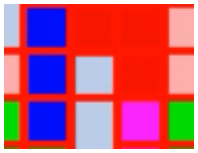

図 16　周囲にブロックが存在することで誤検知が発生してしまっている様子

がかなり増加し，手が付けられない状態となっていくことも分かる．

一方，図 15 から，ブロック 1 個ずつのテンプレート画像（図 4）に基づくルールベースなデバッグ AI では，図 14 の横に 10 個揃ったテンプレート画像を用いた時と比べ，バグ検出の精度が向上されていることが分かる．閾値が 0.90 辺りで正解データにかなり近い状態になっているが，バグを 1 つのみしか判別できなかった Frame#339 の周辺では，閾値が高い故にブロックが検知されず，「ラインが揃ったのに消えない」バグを見逃している．また，図 16 のように，Frame#514 では周囲にブロックが囲まれたことによりブロックが無いにも関わらず，ブロックとして認識し，「ラインが揃ったのに消えない」バグとして誤検知してしまっている．

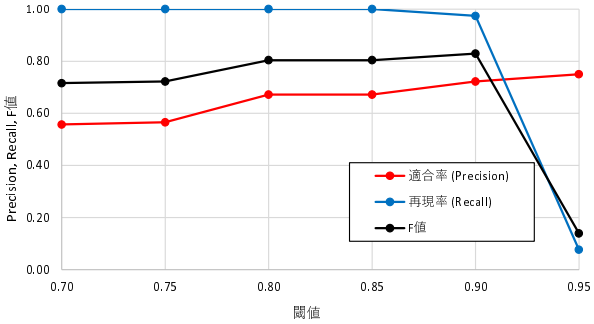

図 17　ブロック 1 個ずつのテンプレート画像の類似度の閾値と平均適合率・再現率・F 値の関係

更に，確率 30 %でバグが混入したプレイ動画を 10 個用意して，ブロック 1 個のテンプレート画像の閾値 0.70,0.75,0.80,0.80,0.85,0.90,0.95 でデバッグを行い，適合率 (Precision)，再現率 (Recall)，F 値を調査した．図 17 にその結果を示す．適合率の最良値は 0.75，再現率の最良値は 1.00，F 値の最良値は 0.83 であった．ここで，閾値を変更して精度を調査しているのは，閾値の設定次第で，精度が変化するかどうかを確認するためである．結果から，閾値が 0.85 から 0.90 の間で最も安定的な精度が得られると分かる．また，混入したバグは 1 つを除き全て検知することができ，再現率が高い結果となったが，適合率が低く，誤検知も多く出てしまう結果となった．閾値が 0.90 の時，ブロック 1 個の検出で所々未検出が見られたため，これ以上閾値を上げてもブロックとして認識しなくなると考えられる．故に，誤検知を減らすためには，ブロックの認識精度を上げる必要があると分かる．

5.2 「ブロックが枠外に移動できてしまう」デバッグ

Frame#432，Frame#535 に「ブロックが枠外に移動できてしまう」バグが発生し，それ以降は残り続けるプレイ動画に対して，ルールベースなゲーム画面認識によるデバッグ AI のプロトタイプの性能を検証すると，ブロック 1 つのテンプレート画像と枠のテンプレート画像の類似度の閾値に依って，図 18 と図 19 が得られた．図 18 は枠のテンプレート画像の閾値を 0.60 に固定して，ブロック 1 つのテンプレート画像の閾値を変化させている．図 19 はブロック 1 つのテンプレート画像の閾値を 0.85 に固定して，枠のテンプレート画像の閾値を変化させている．

まず，図 18 から，Frame#432 の「ブロックが枠外に移動できてしまう」バグは，ブロック 1 つのテンプレート画像の閾値が極端な値でなければ，バグ発見できていることが分かる．しかし，時間経過で発見したバグの数が減少してしまっている．これは，図 10 の 4.2 節で述べた枠外にはみ出るブロックに対する工夫により，枠内に色が残ってしまい，枠として認識しなくなったためと考えられる．Frame#535 の「ブロックが枠外に移動できてしまう」バグは，ブロック 1 つのテンプレート画像の閾値の設定に依らず，検出できなかった．この原因として，図 10 の 4.2 節で述べた枠外にはみ出るブロックに対する工夫が，OpenCV のテンプレートマッチングの仕様上，左から認識

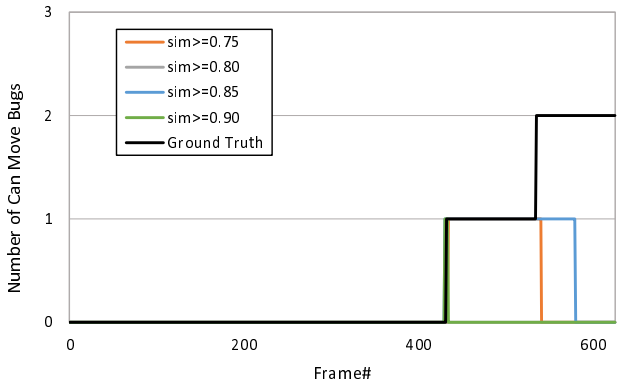

図 18　ブロック 1 個ずつのテンプレート画像（図 4）の閾値に依るルールベースなデバッグ AI のバグ検出数（枠のテンプレート画像（図 7）との類似度の閾値 $sim >= 0.60$ に固定）

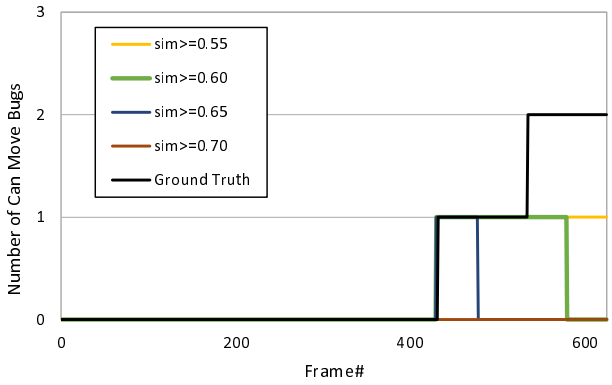

図 19　枠のテンプレート画像（図 7）の閾値に依るルールベースなデバッグ AI のバグ検出数（ブロック 1 個ずつのテンプレート画像（図 4）との類似度の閾値 $sim >= 0.85$ に固定）

するため，左にはみ出たブロックには対応できるが，右にはみ出たブロックには対応できないため，右にはみ出たブロックを認識することができず，バグの見逃しとなったと考えられる．

次に，図 19 では，上述の図 18 についての考察と同様のことが考えられ，枠のテンプレート画像の閾値が 0.55 辺りで最も正解データに近い状態となっている．

更に，確率 30 %でバグが混入したプレイ動画を 10 個用意して，ブロック 1 個のテンプレート画像の閾値 0.70,0.75,0.80,0.85,0.90，枠のテンプレート画像の閾値 0.50,0.55,0.60,0.65 の計 20 通りでデバッグを行い，適合率 (Precision)，再現率 (Recall)，F 値を調査した．図 20 にその結果を示す．適合率の最良値は 1.00，再現率の最良値は 0.88，F 値の最良値は 0.92 であった．但し，正解かどうかを確認する作業は著者（人間）の手によって行われており，一つの場所に重複している検知も一つとしてカウントしている．結果から，発見したバグは 3 つを除き全て正解のバグであり，適合率が高い結果となったが，再現率が低く，未検知も多く出てしまう結果となった．重複している検知も一つとしてカウントしているため，閾値が低いほど検知結果が良くなる傾向にある．しかし

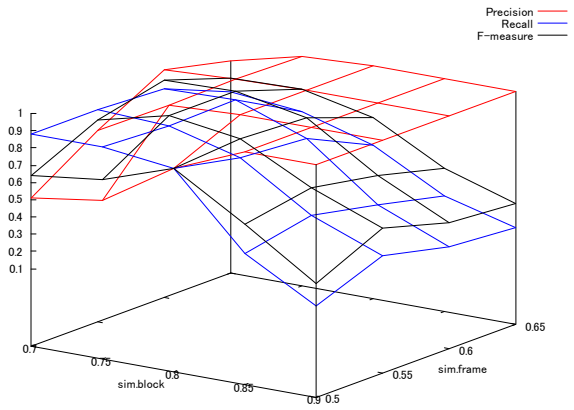

図 20　ブロック 1 個ずつのテンプレート画像の類似度の閾値 $sim.block$ と枠のテンプレート画像の類似度の閾値 $sim.frame$ と平均適合率・再現率・F 値の関係

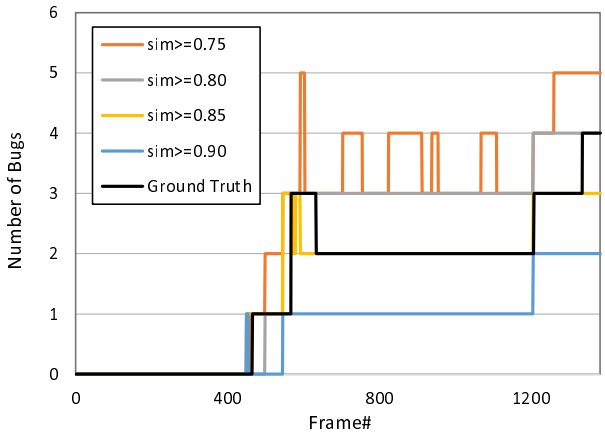

図 21　ブロック 1 個ずつのテンプレート画像（図 4）の閾値に依るルールベースなデバッグ AI のバグ検出数（枠のテンプレート画像（図 7）との類似度の閾値 $sim >= 0.60$ に固定）

ながら，閾値を下げ過ぎると，重複でない誤検知が増加するため，適合率が悪化し，F 値も悪化すると予想される．また，全体を通して，枠外のブロックを完全に検知することができなかったため，バグであると判別されない結果となったと考えられる．故に，未検知を減らすためには，5.1 節のようにブロックの認識精度を上げる必要があると分かる．

5.3 「ラインが揃ったのに消えない」「ブロックが枠外に移動できてしまう」両対応デバッグ

Frame#466，Frame#1336 に「ブロックが枠外に移動できてしまう」バグが発生し，その内の Frame#466 は途中でライン消去により消え，Frame#568 に 2 つ，Frame#1208 に 1 つ「ラインが揃ったのに消えない」バグが発生し，それ以降は残り続けるプレイ動画に対して，ルールベースなゲーム画面認識によるデバッグ AI のプロトタイプの性能を検証すると，ブロック 1 つのテンプレート画像と枠のテンプレート画像の類似度の閾値に依って，図 21 と図 22 が得られた．図 21 は枠のテンプ

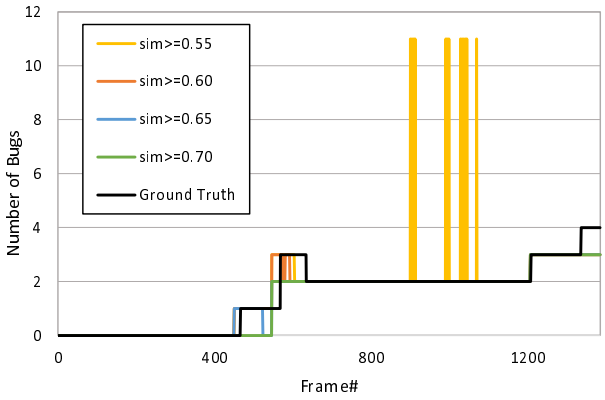

図 22　枠のテンプレート画像（図 7）の閾値に依るルールベースなデバッグ AI のバグ検出数（ブロック 1 個ずつのテンプレート画像（図 4）との類似度の閾値 $sim >= 0.85$ に固定）

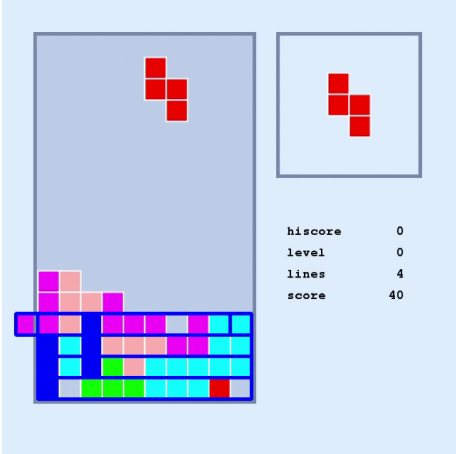

図 23　左にはみ出たブロックから横に 10 個並んでいると誤検知している様子

レート画像の閾値を 0.60 に固定して，ブロック 1 つのテンプレート画像の閾値を変化させている．図 22 はブロック 1 つのテンプレート画像の閾値を 0.85 に固定して，枠のテンプレート画像の閾値を変化させている．

まず，図 21 から，ブロック 1 つのテンプレート画像の閾値が 0.85 のとき，正解データにかなり近い状態となっているが，5.2 節と同様に，右にはみ出るブロックには対応できず，バグを見逃している．また，ブロック 1 つのテンプレート画像の閾値が 0.70,0.75 のとき，図 23 のように，下から 4 行目が，ラインが揃っていないにもかかわらず，左にはみ出たブロックから横に 10 個並んでいると誤検知され，ラインが揃ったと誤認識してしまっており，さらに追加の処理が必要である．

次に，図 22 から，枠のテンプレート画像の閾値が 0.55 から 0.60 の間で，最も正解データに近い状態であると分かる．しかし，枠のテンプレート画像の閾値が 0.55 のとき，かなり多くの誤検知が発生している．これは，枠のテンプレート画像の閾値を低くし過ぎたため，本来枠ではない場所が枠として誤認識され，バグとして検知してしまったためと考えられる．

6　おわりに

本稿では，人間のデバッガーやテスターの代わりとなるデバッグ AI を構築する最初の一歩として，テトリスゲームを題材に，人間プレイヤー操作によるバグ発生を含むプレイ動画を画像認識して，ルールベースで 2 種のバグを発見するデバッグ AI を試作した．その結果，まず「ラインが揃ったのに消えない」バグはより直感的なブロック 10 個横に並んだテンプレート画像よりも，ブロック 1 個ずつのテンプレート画像に基づくマッチングの方が高精度なバグ検出を実現できる可能性が高いことが分かった．また，「ブロックが枠外に移動できてしまう」バグでは，テンプレートマッチングだけでなく，バグ検出の精度を改善するため，Inpainting も用いるなどにより，実現を試みたが，左にはみ出たブロックのみしか検知できないという，精度としては不十分な結果となった．

しかし，適合率，再現率，F 値を 10 個分のバグ動画を用いて調査すると，「ラインが揃ったのに消えない」バグでは，適合率が低いが，再現率が高く，「ブロックが枠外に移動できてしまう」バグでは，適合率が高いが，再現率が低い結果となった．2 つのバグで適合率と再現率を上げるには，ブロック 1 個の認識精度を上げる必要があると分かった．

最後に，本稿の提案手法が他のゲームにも適用可能かどうかについては，テトリスのルールに基づいているので，適用不可能で汎用性が低いと言える．汎用デバッグ AI を構築するには，本稿では用いなかったが，機械学習を用いて教師付き学習をさせることによって，幅広いバグに対応させる手法が考えられる．

今後の研究課題としては，本稿で用いた OpenCV のテンプレートマッチングや Inpainting をより精度の高いもので代用し，精度の向上を図ったり，出来る限り網羅的な種類のバグをルールベースで検出可能か検証していく．また，未知のバグへの対応のため，機械学習や深層学習なども検討していく．

文　献

[1] 日経 XTREND，“世界ゲーム市場は約 22 兆円に　国内市場はゲームアプリが 1.3 兆円，” https://xtrend.nikkei.com/atcl/contents/watch/00013/01964/ (参照 2022-12-26).
[2] 茂原 敦之，水口 充，“汎用ゲームエンジンに外付けするデバッグ環境の提案，” エンタテインメントコンピューティングシンポジウム 2018 論文集，pp.31–35 (2018).
[3] 髙橋 秀太朗，服部 峻，高原 まどか，“テトリスのためのルールベースなゲーム画面認識によるデバッグ AI の試作，” 日本デジタルゲーム学会 (DiGRA) 2022 年夏季研究発表大会 予稿集，pp.45–48 (2022).
[4] 高田 亮介，橋本 剛，“無限 1UP を題材としたアクションゲームの裏技を発見する自己学習手法の提案，” 情報処理学会研究報告ゲーム情報学 (GI)，Vol.2018，No.5，pp.1–7 (2018).
[5] 清木 昌，“ゲームシナリオのモデル検査，” 情報処理学会研究報告ゲーム情報学 (GI)，Vol.2004，No.28，pp.51–56 (2004).
[6] 木間 塚達，野田 夏子，“モデル検査によるゲームシナリオの不具合発見ツールの提案，” 情報処理学会研究報告ソフトウェア工学 (SE)，Vol.2017，No.13，pp.1–9 (2017).
[7] Rosetta Code，“Tetris，” https://rosettacode.org/wiki/Tetris (参照 2023-1-10).