CapStudio: プログラムの実行画面の録画映像を利用した統合開発環境

深堀 孔明　　坂本 大介　　加藤 淳　　五十嵐 健夫 *

概要. 通常，プログラム開発は，コードエディタ上でプログラムのソースコードを編集することによって行われる．しかし，ゲーム開発においては，キャラクタ画像やボタンの配置など見た目に関わる部分はゲームの実行画面を見ながらリアルタイムに編集できた方が直感的である．ゲームエンジンのシーンエディタといった，プレビュー画面上でパラメータを編集できるシステムもあるが，フレームワーク上で決められた制約があり，ゲーム画面の描画などのプリミティブな処理をプログラマが編集できず，コーディングの自由度が低くなるという問題がある．そこで，本稿ではゲーム画面の録画映像をベースとしたエディタ（以下スクリーンキャスト）を提案する．これをコードエディタと連携させることで，プログラマはプログラム全体をコードエディタで記述しつつも，ゲーム画面のデザインに関わる定数値やリソースファイルをスクリーンキャスト上でリアルタイムに編集できる．具体的には，定数値やリソースが編集されると即座にその結果がスクリーンキャストに反映される．反対に，スクリーンキャストのゲームオブジェクトから対応するコード片やリソースファイルへアクセスしたり，ゲームオブジェクトの位置やサイズを直接操作してソースコード中の定数値を調整できる．本稿では，スクリーンキャストを組み込んだ統合開発環境 CapStudio を実装し，本手法の可用性を検証した．

1　はじめに

ゲーム開発など，ビジュアルな要素を多く使用するプログラミング開発においては，キャラクタ画像やボタンの配置といった，見た目に関わる部分の調整は非常に重要である．キャラクタ画像は配色を微妙に変えるだけで大きく印象が変わるし，ボタンなどの GUI 部品の配置方法によってゲームの操作性は大きく変化する．しかし従来のコードベースの開発環境においては，このようなビジュアル要素の開発を支援する枠組みはほとんどなかった．このため，たとえばプログラムの編集中にはゲーム画面は表示されないため，キャラクタ画像やボタンの表示位置を修正するたびにプログラム全体をビルド・実行してゲーム画面を表示し，見栄えを確認する必要があった．

一方，Unity [9] などの近代的なゲームエンジンでは，プログラムの編集中にゲームのプレビュー画面が常に表示される．プログラマはパラメータの修正結果をリアルタイムに確認したり，画面上のゲームオブジェクトを直接動かすことで位置パラメータを調節することができる．しかし，これらのシステムにはフレームワーク上で決められた制約があり，ゲーム画面の描画といったプリミティブな処理をプログラマが編集することができず，コードベースの開発環境に比べてコーディングの自由度が下がってしまうという問題がある．

本稿では，従来のコードベースの開発環境に対してゲームエンジンのような対話可能なプレビュー画面を提供することを目的とする．そのためのシステムとして，対話可能なゲーム画面の録画映像であるスクリーンキャストを組み込んだ統合開発環境 CapStudio を提案する (図 1)．プログラムの実行中，CapStudio はゲーム画面をバックグラウンドで録画する．実行後，録画映像はスクリーンキャストとして表示される．スクリーンキャストは一般的な動画プレイヤと同様に，映像を再生したり，再生時刻をシークバーで変更できる．

ただし，実際にシステムが記録する情報は，ゲームオブジェクトの描画位置や大きさといった描画履歴であり，システムはこれを用いて録画映像を生成している．また，この描画履歴を用いることで，スクリーンキャストとコードエディタを協調して動作させることができる．具体的には，プログラマがソースコードや画像などのリソースファイルを編集する

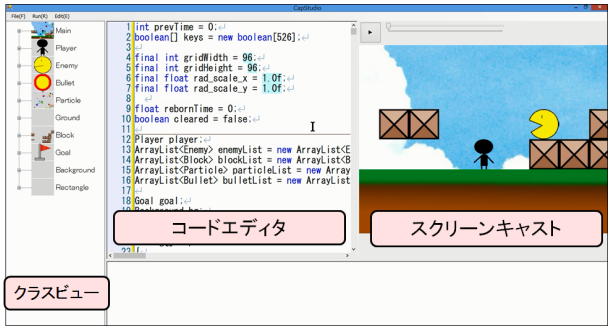

図 1. CapStudio のインタフェース.

* Koumei Fukahori, Daisuke Sakamoto, Jun Kato, Takeo Igarashi, 東京大学

と，即座に編集結果がスクリーンキャストに反映される．反対に，スクリーンキャスト上のゲームオブジェクトを選択することで，対応するコード片やリソースファイルへアクセスできる．さらに，ゲームオブジェクトの位置や大きさを直接操作して，ソースコード中の対応する定数値を書きかえることができる．本稿では作成したシステムについて述べ，その実装を説明したあと，3つのサンプルプログラムを用いて本システムの可用性を議論する．

2 関連研究

本研究は 2012 年の Bret Victor によるデモビデオ“Inventing on Principle”[1] に強く触発されている．このビデオでは，本システムのようにプログラムの実行画面を開発環境に埋め込み，コードエディタと連携させる手法が紹介されている．ただし，これはあくまでコンセプトビデオであり，システムの具体的な実装については言及されていない．一方で，本稿では，このような実行画面とコードエディタの連携機能を，スクリーンキャストを用いることで実現し，その具体的な実装を示している．

実行画面の録画映像をベースとしたインタフェースを持つシステムとして，Whyline for Java [5] というデバッグツールがある．Whyline は実行時に記録した実行履歴を解析し，プログラムの挙動の原因となるコード片を特定する．たとえば，プログラマが録画映像上のゲームオブジェクトを選択することで，そのオブジェクトの色や位置を決定したコード位置へジャンプすることができる．しかし，Whyline は本システムとは違い，ソースコードの編集を目的としたツールではない．そのため，コードの編集結果を即座に録画映像に反映したり，映像内のゲームオブジェクトを動かしてソースコード中のパラメータを書きかえることはできない．

プログラムを明示的に再実行せずにソースコードの修正結果を計算する手法として，バックグラウンドでプログラム全体を再ビルド・再実行して再計算を行う方法がある [3, 4]．しかしこの方法は，外部プロセスと通信を行うプログラムを開発する場合，暗黙的な再ビルド・再実行によって，意図せずに外部プロセスの状態を変えてしまう恐れがある．一方，本稿で提案する手法では必要な情報をすべて実行時に記録し，記録した情報のみを用いて再計算を行うため，予期しない外部プロセス・リソースの改変は起きない．また, プログラムの編集状態と実行状態を区別しない Live Programming 言語 [2, 6, 8] も，コード編集の結果を再実行せずに実行画面に反映することができる．しかし，上記の手法と同様に、プログラムの編集中に予期せず外部プロセスに影響をおよぼす恐れがある。

また，本システムともっとも関連のあるシステムとして，Unity [9] や Unreal Engine [10] といったゲームエンジンのシーンエディタや Visual Studio [11] などの GUI エディタがある．これらの開発環境では，プログラムの編集中，実行画面のプレビューが常に表示され，パラメータの修正結果をリアルタイムで反映したり，プレビュー画面上のゲームオブジェクトを直接動かして位置パラメータを調節できる．しかし，これらのシステムにはフレームワーク上で決められた制約があり，ゲーム画面の描画といったプリミティブな処理をプログラマが編集できないため，コードベースの開発環境に比べてコーディングの自由度が下がってしまうという問題がある．また, プレビュー画面内の時間の流れは不可逆で，本システムのスクリーンキャストと違い，シークバーを動かして時間を巻き戻すことはできない．

3 提案システム CapStudio

CapStudio の実行画面を図 1 に示す．プログラマはまずコードエディタにソースコードを記述し，その後，スクリーンキャストを利用してゲーム画面のデザインを修正する．

提案手法の最大のポイントは，ゲームエンジンのシーンエディタのようなパラメータ調整用のエディタを，コードベースの開発環境に対して提供することにある．一般に，描画処理を含めたすべてのソースコードがプログラマによって記述される場合，プログラムの挙動を静的に予測してプレビュー画面を生成することはできない．そこで本研究では，実行時に出力されたゲーム画面を擬似的なプレビュー画面に見立てる．まず，システムはゲーム画面の描画履歴を実行時に記録し，これを利用してゲーム画面の録画映像をスクリーンキャストとして生成する．さらに，記録した描画履歴を利用してスクリーンキャストとコード・リソース間の連携機能を実現する．

3.1 スクリーンキャストの生成

プログラマはまず，コードエディタに Processing 言語 [7] でソースコードを記述する．CapStudio のコードエディタは公式の開発環境 Processing IDE（PDE）と互換性がある．PDE で実行できるコードは CapStudio でも実行でき，その逆も真である．メニューバーの実行ボタンを押すと，プログラムがビルド・実行されゲーム画面が表示される．

プログラムの実行時，システムはバックグラウンドでゲーム画面の描画に関わった関数 (以下，描画関数) の呼び出し履歴を記録する．たとえば，図 2 のようにゲーム画面上に画像がひとつだけ表示されるプログラムでは，ゲーム画面の大きさを指定する *size()*, 画像データをファイルから読み込む *loadImage()*, 画面をクリアする *background()*, 画像を描画する *image()* のいずれかが呼び出されるたびに，そのコード位置，返り値，仮引数の式に含まれる変数の値といった関数呼び出し情報を記録する．これによ

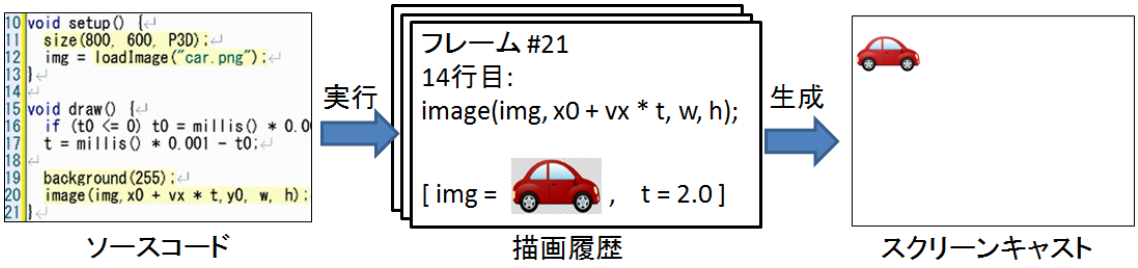

図 2. スクリーンキャストの生成

り，描画されたゲームオブジェクトの位置や大きさ，およびそれを描画したコードが特定できる．ただし，ゲーム画面の描画には Processig が公式に提供している API 関数だけが使われると仮定している．つまり，ソースコード中のどの関数が描画に関わっていて，どの引数が位置や大きさを指定するのかという API の仕様はシステムがあらかじめわかっている．もし OpenGL や Java Swing など別の描画 API を使う場合は，各 API 関数の仕様を別途システムに組み込む必要がある．

プログラムの実行が終了すると，システムは記録された描画履歴から，ゲーム画面の録画映像がスクリーンキャストとして生成される (図3)．スクリーンキャストは動画プレイヤのように，再生ボタン（▷）を押して録画映像を再生したり，シークバーを動かして再生時刻を変更できる．

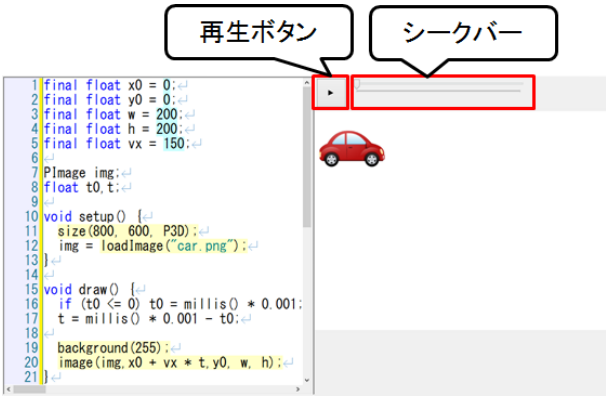

図 3. CapStudio で開発したプログラムの例．車の画像が水平方向に等速移動する．

3.2 コード・リソース編集結果のスクリーンキャストへの即時反映

プログラムの実行後，システムは，ゲーム画面のデザインに関わるソースコードをハイライトする．ゲーム画面のデザインに関わるソースコードとは，本稿では，1) 描画関数呼び出し文の引数，2) 定数宣言式の右辺，の 2 つを指す．たとえば，図 3 のコードでは，*loadImage()*，*image()* といった描画関数の引数と，x0，y0 といった定数の値がハイライトされる．プログラマがこれらのいずれかを書きかえると，システムは再度ソースコードの構文解析を行い，新しい引数式・定数値および前節で記録した描画履歴にしたがってスクリーンキャストを再描画する (図 4-a)．同様に，プログラマが画像などのリソースファイルを編集すると，システムは編集されたリソースを読み込み直して，スクリーンキャストを再描画する (図 4-b)．

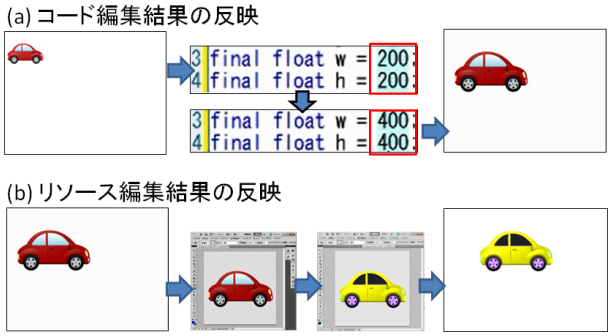

図 4. コード・リソース編集結果のスクリーンキャストへの反映．

3.3 スクリーンキャストを用いたコード・リソースの編集

スクリーンキャスト内のゲームオブジェクトはすべてクリッカブルであり，クリックすることで対応するコードやリソースへアクセスできる．左クリックすると，それを描画した関数呼び出しが赤くハイライトされ，そこへカーソルが移動する (図 5-a)．右クリックすると，対応する画像ファイルを開くためのメニューが表示され，選択した画像ファイルが外部の画像編集ソフトで開かれる (図 5-b)．前述したように，これらのコードや画像ファイルが編集されると，即座に結果がスクリーンキャストに反映される．

また，スクリーンキャストからコード内の定数値を調節できる．スクリーンキャスト上のすべてのゲームオブジェクトは位置や大きさをドラッグ操作で変更できる (図 6-a)．こうしてゲームオブジェクトの位置・

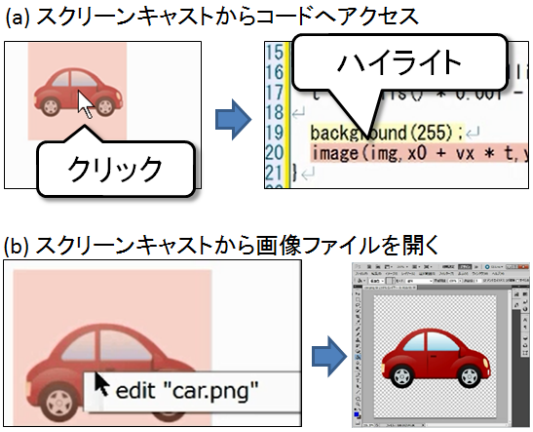

図 5. スクリーンキャストからコード・リソースへのアクセス

大きさを変更すると，同様の結果になるようにソースコードが自動で書きかえられる (図 6-b)．デフォルトでは対応する関数呼び出し文の引数に定数値が加えられる．書きかえられた引数を右クリックすると，別の書きかえパターンが列挙される (図 6-c)．列挙された各パターンは，引数の式に登場する定数を書きかえ対象とする．たとえば，*image(img,x0+vx*t,y,w,h)* という関数呼び出しによって描画された画像を，左端の X 座標が 150 になるように移動したとする．ただし，x0，vx, t の値はそれぞれ 50，25, 2 で x0, vx が定数である．このとき，まず第二引数に目的値 (150) と現在の値 (100) との差分 (50) が加算され，*rect(img,x0+vx*t+50,y,w,h)* と書きかえられる．その後，式 *x0+vx*t+50* が右クリックされると，定数 x0, vx を書きかえ対象としたパターンがそれぞれ「*final float x0 = 100*」「*final float vx = 50*」と表示される．たとえば前者のパターンを選択すると，第二引数が書きかえ前の式 *x0+vx*t* に戻り，x0 の宣言式が *final float x0 = 100* に書きかえられる．

ただし，引数に現れる定数の内，次の 2 つの条件を満たすものが書きかえ対象となる．1) 描画関数の引数のみに影響を与える．2) 定数がどんな値になるべきか計算可能．たとえば if 文の条件式に使われる定数は書きかえ対象にはならない．また，現在の実装では，引数式が定数に関する線形式の形になっている場合のみ目標値を計算できる．つまり，引数の式が a * c1 + b / c2 で c1，c2 が定数だとすると，c1 のみが書きかえ対象となる．

4 実装

CapStudio は Processing 2.0 beta 8 の開発環境である．インタフェース部分は C#の WinForm アプリケーションとして実装されており，Windows 7 および Windows 8 で動作している．以下，前章で

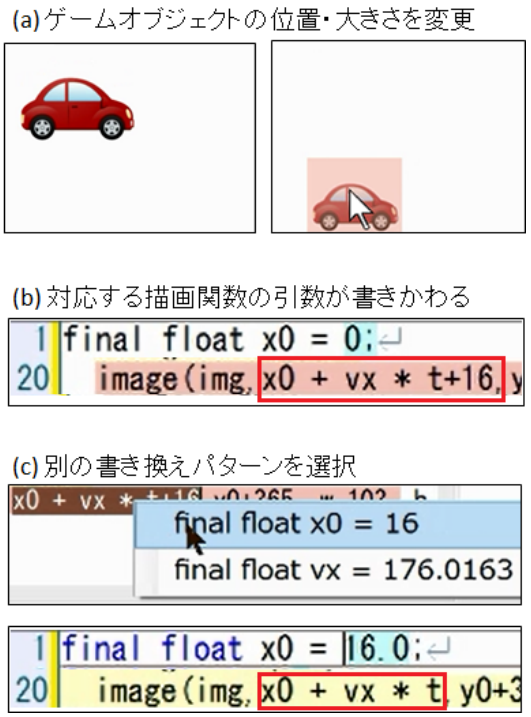

図 6. スクリーンキャストを用いたコード編集

紹介した機能について，実装アルゴリズムを述べる．

4.1 描画履歴の記録・スクリーンキャストの生成

実行時に描画履歴を記録するため，プログラムがビルドされる前に，描画関数の呼び出し文に対してコード書きかえが行われる．ソースコード中の描画関数呼び出し文の前後に，その呼び出し文のコード位置，呼び出し時の返り値および引数の式に登場する変数の値を記録するコードが自動挿入される．

記録された描画履歴を用いて，現在シークバーが示している時点（フレーム）のゲーム画面を再現する．まず，プログラムの実行が終わると，システムは実行時に描画関数によって読み込まれたリソースファイルをすべて読み込む．次に，コードエディタに記述されたソースコードを構文解析し，定数の値と描画関数の引数式を抽出する．そして，実行時に描画関数が呼び出されたときの引数値を，引数式に定数値と実行時に記録された変数値を代入することで求める．その後システムは，読み込んだリソースと求めた引数値を使って，目的のフレームの間に呼び出された描画関数を順に再実行することで，スクリーンキャストを描画する．

4.2 コード・リソース編集結果のスクリーンキャストへの即時反映

プログラマがコードやリソースファイルを書きかえると，スクリーンキャストは実行時に読み込まれたリソースファイルをすべて読み込み直す．その後，

ソースコードの構文解析を再実行し，定数の値と，描画関数の引数式を更新する．その後，新しい定数値・引数式を用いて，各時点における描画関数の引数値を再計算する．その後，現在のフレームで呼び出された描画関数を順に実行しなおして，スクリーンキャストを再描画する．

4.3 スクリーンキャストを用いたコード・リソースの編集

プログラマがスクリーンキャストをクリックすると，現在のフレームで発行された描画関数の呼び出しをすべて走査し，マウスカーソルと接触しているゲームオブジェクトを探す．見つかったら，それを描画した関数呼び出し文をハイライトする．あるいは引数にリソースオブジェクト (PImage 型の画像オブエジェクトなど) が与えられている場合は，その元ファイルのファイルパスを表示する．

同様に，プログラマがスクリーンキャスト上のゲームオブジェクトをドラッグすると，まずすべての描画関数呼び出しを走査して，ドラッグされたゲームオブジェクトを探す．次に，マウスの移動量に応じてゲームオブジェクトの新しい座標・大きさを求める．その後，その位置・大きさで描画されるように対応する描画関数の引数の値を求め，現在の引数の値との差分を引数式に加算する．同時に，別の書きかえパターンについて，書きかえ対象とする定数の目標値を，その定数に関する一次方程式を解くことで求める．

4.4 スクリーンキャスト生成手法の応用

２章で解説したコード・リソース編集のための機能に加えて，スクリーンキャストの生成手法を応用して，次の３つの機能を実装している (図 7)．1) あるクラスのメンバ関数の中で呼び出された描画関数のみを選択して再実行することで，そのクラスのサムネイル画像を生成する．2) 他のフレームで描画されるゲームオブジェクトを現在のフレームに重ねて描画することで，ゲームオブジェクトの軌跡を残像として表示する．さらに，3) ゲームオブジェクトの描画位置をずらすことで，視点を二次元的に移動させ，画面外に描画されていたゲームオブジェクトを表示させることができる．

5 サンプルプログラム

作成したシステムの可用性を検討するため３つのサンプルプログラムを作成した (図 8)．１つ目の例ではゲーム画面のレイアウト，２つ目の例ではリソース編集における，スクリーンキャストの有用性を議論する．３つ目の例では，実行時に外部プロセス・リソースと通信するプログラムの開発において，本システムが有効な性質を持つことを示す．

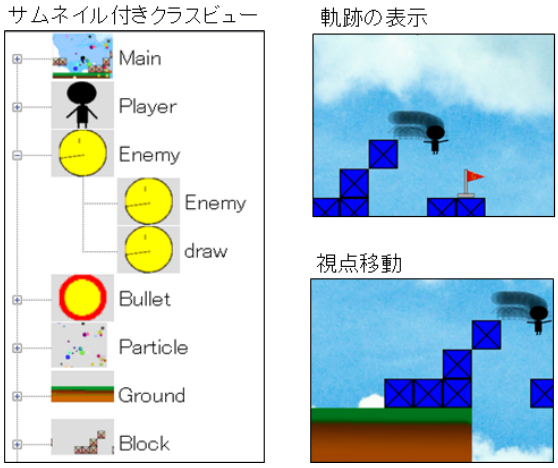

図 7. スクリーンキャスト生成手法の応用

5.1 ゲームのタイトル画面

図 8-a はゲームのタイトル画面に模した GUI プログラムである．タイトル・メニュー項目がそれぞれラベル・ボタンとして表示され，それぞれの位置と大きさはソースコード中で定数として指定されている．これらのオブジェクトを一度実行時に表示させてしまえば，あとはスクリーンキャストでオブジェクトをドラッグして位置・大きさを調節できる．もちろん，既存 GUI エディタやゲームエンジンでも，このように GUI 部品のレイアウトを直接操作することはできる．しかし本研究では，ソースコードがすべてプログラマによって書かれたプログラムに対して同様の操作を可能にしたことが重要であると考えている．

5.2 2D アクションゲーム

図 8-b は簡単な 2D 横スクロールアクションゲームである．ユーザはキャラクタを操作し，道中の敵を弾丸で倒しながらゴールを目指す．キャラクタや敵といったゲームオブジェクトはすべて，外部ファイルから読み込まれた画像を使って描画される．これらの画像を編集するためには，通常はファイルマネージャから画像を保存しているフォルダへ移動し，目的の画像を開く必要がある．一方，本システムでは，スクリーンキャスト上のゲームオブジェクトから直接目的の画像を開くことができ，ユーザにとっての負荷が大きく軽減されている．特に，弾丸のように一瞬しか現れないゲームオブジェクトは，再生時間を巻き戻せない映像内で捉えることは難しく，ゲームエンジンのシーンエディタを利用した場合でも，ゲームオブジェクトから画像を直接開くことは難しかった．しかし本システムでは，シークバーを動かして再生時刻を調整することで，そのようなオブジェクトもスクリーンキャスト上で簡単に捉えることができる．

図 8. サンプルアプリケーション.

5.3 ネットワーククライアント

図 8-c はネットワーク通信を行う簡単なメディアアート作品である．Kinect から取得された骨格データを用いて，人間の手からパーティクルを放出する．ただし，Kinect の公式 SDK は Processing に対応していないので，骨格データは外部の C#プロセスに取得させ，TCP 通信で Processing プロセスへ送信している．プログラムの実行を終了すると，C#プロセスとのネットワーク接続が遮断され，Processing プロセスは新たな人間の骨格データを取得できなくなる．このとき，関連研究の章で紹介したバックグラウンドでプログラム全体を再ビルド·再実行する手法では，プログラマがソースコードを編集したときに正しく編集結果を計算できない．一方本システムは，実行時に骨格データの情報が描画履歴として記録されており，これを使って計算を行うため，コードの編集結果がスクリーンキャストに正しく反映される．

6 まとめと今後の課題

本稿では，対話可能なプログラムの録画映像であるスクリーンキャストを組み込んだゲーム開発のための統合開発環境 CapStudio を提案した．スクリーンキャストは従来のコードエディタと協調して動作し，プログラマは全てのプログラムコードをコードエディタで記述しつつも，ゲーム画面のデザインをスクリーンキャスト上で直感的に修正できる．

本提案の問題としては，スクリーンキャストがソースコードの特定箇所の変更結果しか反映できないことが挙げられる．現状の実現方法では，描画関数の呼び出し文と定数の定義式の変化のみトラッキングしている．したがって，ゲームオブジェクト同士の衝突判定といったロジックに関わるコードを書きかえても，スクリーンキャストには反映されない．本システムはあくまでゲームオブジェクトの配置や色合いといった，画面デザインの修正を支援することを目的としている．

現時点では，Processing の描画 API の内，特に現在は，画像読み込み·画像描画といった，2D ゲームの開発に必要な描画関数のみに対応している．したがって今後は，3D モデルの描画を含めた Processing の描画 API のフルセットへの対応を予定している．それと並行して，スクリーンキャスト上のゲームオブジェクトを三次元的に操作するためのインタフェースも実装する．さらに，OpenGL や Java Swing といった，より低レベルな描画 API への対応も検討している．

参考文献

[1] Bret Victor - Inventing on Principle. http://vimeo.com/36579366.

[2] J. Edwards. Subtext: Uncovering the Simplicity of Programming. In *Proc. OOPSLA '05*, pp. 505–518, 2005.

[3] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST '08*, pp. 91–100, 2008.

[4] J. Kato, S. McDirmid, and X. Cao. DejaVu: integrated support for developing interactive camera-based programs. In *Proc. UIST '12*, pp. 189–196, 2012.

[5] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. ICSE '08*, pp. 301–310, 2008.

[6] S. McDirmid. Living it up with a live programming language. In *Proc. OOPSLA '07*, pp. 623–638, 2007.

[7] Processsing. http://processing.org.

[8] Welcome to Self - Self - the power of simplicity. http://selflanguage.org/.

[9] Unity3D Game Engine. http://unity3d.com.

[10] Game Engine Technology by Unreal. http://www.unrealengine.com/.

[11] Microsoft Visual Studio. http://msdn.microsoft.com/ja-jp/vstudio.