社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

GameFrameWork Cerium における Sequential な Game Program の分割と動作の検証

小林　佑亮†　多賀野海人†　金城　裕†　河野　真治†

† 琉球大学 理工学研究科 情報工学専攻 並列信頼研究室

あらまし　本稿では Sequential なゲームプログラムを 逐次実行した時と、Task に分割し実行した時の動作が同一であることを確認する為のテスト手法を提案する。ランダム要素であるプレイヤー入力と乱数の固定化を行うことでバグの再現性を向上させ、さらに高速にテストを行える環境を構築する。
キーワード　ゲーム テスト Cell Cerium

Division and verification of Sequential Game Program on GameFrameWork Cerium

Yusuke KOBAYASHI†, Kaito TAGANO†, Yutaka KINJO†, and Shinji KONO†

† Concurrency Reliance Laboratory, Information Engineering Course, Faculty of Engineering Graduate School of Engineering and Science, University of the Ryukyus.

Abstract　We test divided game program. We immobilize random element that player input and random numbers, and check behavior of divided program and sequential program. At that we constract fast testing environment.
Key words　game test Cell Cerium

1. はじめに

我々は、これまで家庭用ゲーム機上におけるゲームプログラミングをサポートするオープンな開発フレームワークの研究を行ってきた。現在は PlayStation3 上での開発を行っている。

PlayStation3 のアーキテクチャは Cell Broadband Engine と呼ばれ、複数の CPU で構成される。我々は、このような Many Core Architecture を用いた並列プログラムの開発フレームワークとして Cerium Game Engine を開発した。Cerium では、プログラムを Task という単位に分割し、これを各 CPU に送ることで並列実行を実現している。

Cerium はゲームプログラムをサポートしたフレームワークである。ゲームプログラムの特徴としては、プレイヤーのゲームパッドからの入力やコード内に埋め込まれた乱数などの非決定的な要素が多く、バグの再現性が低いことが挙げられる。

また、Cerium におけるゲーム開発ではプログラムを Task に分割するが、Task 間でのパラメータの同期や Task 処理の実行順序によって単純にシーケンシャルに書かれたゲームプログラムを Task に分割して処理させても、元のプログラムを逐次実行させた時と同じ動作をすることは保証されない。

そこで本研究ではシーケンシャルなゲームプログラムと Task に分割したゲームプログラムの動作が同一であることを確認するためのテスト手法を提案する。シーケンシャルに書かれたゲームプログラムとそれを Task に分割したゲームプログラムをテストモデルとし、プレイヤー入力や乱数などの非決定的な要素の固定化により、バグの再現性を保つ。

2. Cell BE と Cerium

2.1 Cell Broadband Engine

Cell Broadband Engine は SCEI と IBM によって開発された CPU である。2 thread の PPE(PowerPC Processor Element) と、8 個の SPE (Synergistic Processor Element) からなる非対称なマルチコアプロセッサであり、高速リングパスである EIB(Element Interface Bus) で構成されている。Cerium の動作環境である PS3 Linux では PPE と 6 個の SPE が使用できる。(図 1)

2.2 Game Framework Cerium

Cerium は我々が提案したゲーム開発フレームワークで、独自の Rendering Engine を持つ。Cerium は C++ で実装されており、画像の読み込みや入力デバイスは SDL を用いて行っている。以下に Cerium を構成する 3 つの機能を列挙する。

SceneGraph:　オブジェクトのパラメータやポリゴン情報を tree 構造のノードで管理する

RenderingEngine:　3 種類の Task によって並列に描画処理

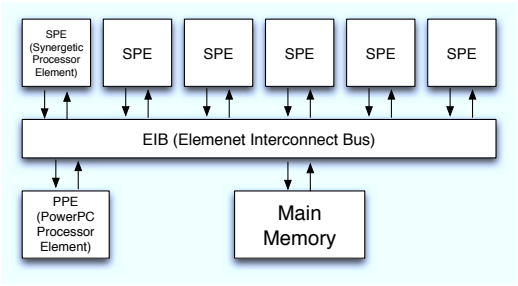

図 1　Cell Broadband Engine

表 1　Task Manager の API

create_task	Task を生成する
run	実行 Task Queue の実行
set_inData	Task への入力データのアドレスを追加
set_outData	Task からのデータ出力先アドレスを追加
set_param	Task に 32 bit の情報を追加
wait_for	Task 同士の依存関係をセット
set_cpu	Task を実行する CPU(PPE,SPE0〜5) の設定
set_post	Task が終了した後 PPE 側で実行される関数の登録

を行う

TaskManager:　Task を動的に各 CPU (PPE,SPE) に割り振る

2.3　Cerium の Task

Cerium で使用される Task への分割単位はサブルーチンまたは関数としている。Task には様々な API が用意されており、実行する CPU の選択や入出力されるデータの設定、Task の依存関係などがセット出来るようになっている。

3.　ゲームプログラムのテストの特徴

多くのゲームでは多数のオブジェクトが存在し、プレイヤーのコントローラー入力やゲームの進行状況によって新たなオブジェクトが生成される。生成されたオブジェクトは他のオブジェクトの座標などのパラメータに影響され、衝突判定を行ったり、挙動が変化する。

この為、ゲームプログラムでは遷移する状態が膨大であり、一般的なテスト駆動のように遷移する状態が仕様の範囲内に収まるのかチェックするようなテストは向かない。ゲームプログラムは実際にプレイヤーがゲームをプレイすることが重要なテストとなる。

3.1　プレイヤーの入力の不定性

プレイヤーの入力は常に非決定的であり、例え同じ人間が同じゲームの同じ場面をプレイしたとしても毎回全く同じ入力をする可能性は極めて低い。こうした事からプレイヤーは制御不能なランダム要素であると考えられ、ゲームプログラムテストにおけるバグの再現性を低下させている。

3.2　ゲームにおける乱数

ゲームにおける乱数は、オブジェクトの振る舞いに多様性を持たせたり、ランダムな配置を実現する為に使われ、ゲームのボリュームや面白さを広げる役割を担ってきた。予測不能な乱数はゲームプレイにおいては面白さを牽引する要因となるが、テスト環境では、こうした乱数のランダム性はデバッグをする上でバグの再現を困難にする。乱数生成器を無効にするか、定数でシードすることによってバグの再現性を下げることなく、テストすることが出来る

3.3　SPE における乱数生成の非決定性

通常のシーケンシャルなプログラムでは、乱数を必要とする処理系が 1 つの乱数列から順番に乱数を取得し、使用する。しかし並列プログラムではこの処理系は Task として SPE に送られる。乱数列は SPE 毎に独自に生成されるため、各 Task が受け取る乱数は逐次実行した時とは異なる値となってしまう。また、SPE 内でも Task 同士に依存関係を持たせない限り、Task の実行順序が保証されないのでこちらも受け取る乱数が不定となる原因となる。(図 2)

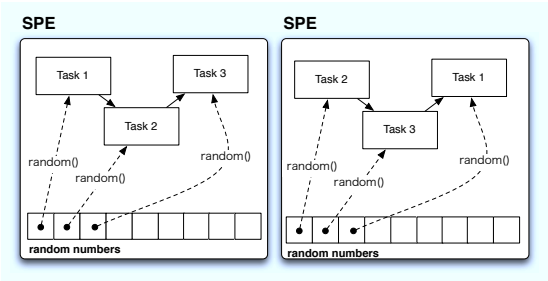

図 2　SPE 内での乱数の生成

4.　テストモデルとなるシューティングゲーム

4.1　Super Dandy

Super Dandy は我々が PlayStation でのゲーム開発を行っていた 1998 年に開発されたシューティングゲームである。SuperDandy は開発する環境が変わる度に移植されており、過去には PlayStation2 Linux、OpenGL バージョンも作られた。(図 3)

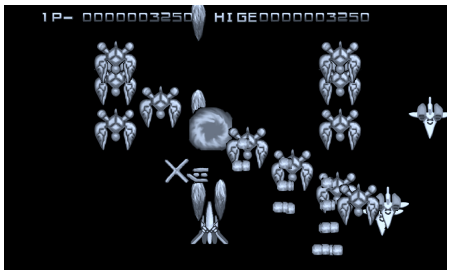

図 3　Super Dandy

Super Dandy が伝統的に移植されてきた背景には、全 5 ステージのある程度のボリュームのあるゲームであること、衝突判定やオブジェクト管理、ステージクリアによるシーン切り替えなど、基本的なゲームとしての要素が入っていること、そして動作結果を過去の環境と比較することで新たな環境のチューニングが行えることが挙げられる。

4.2　Task Dandy(Super Dandy Task version)

本研究を進めるにあたり、Super Dandy を Cerium の Task

で書き換えた Task Dandy を作成した。Task Dandy はできるだけ元の Super Dandy のコードやデータ構造を流用し、比較、テストが容易に行えるように設計した。

並列実行には Amdahl 則があり、使用する CPU を増やしても、元のプログラムの並列化率が低ければその性能を活かすことはできない。この為、Super Dandy において処理の大部分を占めているオブジェクトの動作 (Move) と 衝突判定 (Collision) を Task Dandy では Task として書き直している。

Super Dandy では Move や Collision は state_update() や collision_detect() という関数で行っている。Task Dandy では、この処理の代わりに Move Task や Collision Task を生成している。また、obj_draw() はオブジェクトの描画を行う関数であったが、Task Dandy では SceneGraph の tree を生成している。ゲームの処理を抜けると、さきほど生成した SceneGraph の tree から描画処理を行う 3 つの Task を生成する。(図 4)

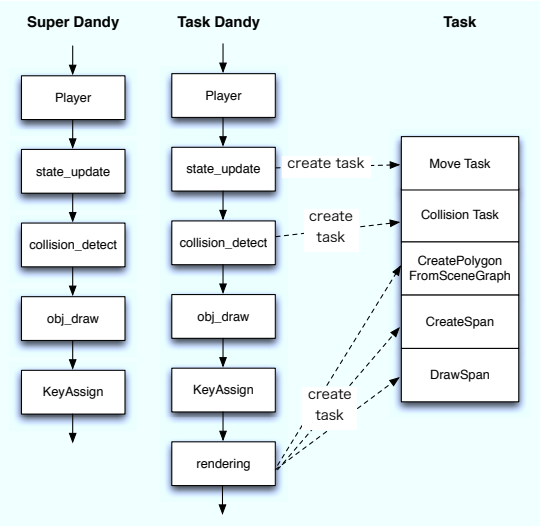

図 4　Super Dandy と Task Dandy の処理

5.　テスト環境の構築

5.1　テストログに記述する情報とそのタイミング

シーケンシャルなプログラムを Task に分割した際に、新たに発生するバグとして、本研究では以下の項目に焦点を当てた。

- Task 間のデータの同期
- Task の実行順序
- Task の定義

このうち、Task の定義については、Task の中身が非常に小さい為 (Super Dandy なら 20〜100 行程度)、Task の inData や outData を調べるといった従来のテスト手法でも十分にテストが可能である。その他の 2 つについては、いずれも衝突判定の際に生じるバグである。(図 5)

そこで、オブジェクトが被弾した時、そしてオブジェクトが生成された時にテストログを取ることで効率的にバグを発見することができると考えた。以下に実際に収集したテストログの例を示す。

```
F64: CREATE  [NAME]enemy_greenclab_0
[COORD]x= 120.000000  y= -128.000000

F85: DELETE  [NAME]enemy_greenclab_0
[COORD]x= 120.000000  y= -44.000000
       vx= 0.000000  vy= 4.000000
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0
```

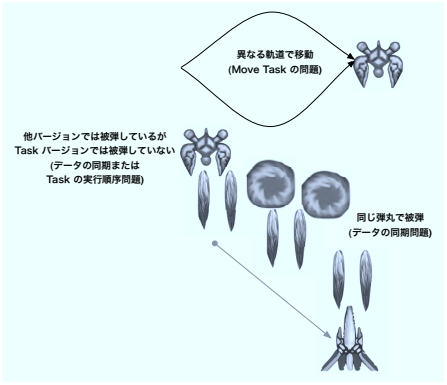

図 5　Task Dandy で起こりうるバグ

それぞれのパラメータの詳細は次のとおりである。

- **F64,F85:** 生成、もしくは被弾した時の経過フレーム。
- **CREATE,DELETE:** CREATE ならオブジェクトが生成された、DELETE ならオブジェクトが被弾した事を表す。
- **NAME:** オブジェクトの種類と ID。ID はオブジェクトの種類毎に 0 から順番に付けられるのでどのオブジェクトの情報なのかを特定できる。
- **COORD:** オブジェクトの xy 座標と xy 方向の速度。
- **BULLET:** その瞬間に画面内に存在した弾の数。差異があれば同期が取れていないことを示す。

これにより、フレーム単位でどのオブジェクトが生成、または被弾したのか知ることができる。trace モードでプレイヤーの入力を固定し、各バージョンでテストログを取り、その差異を調べることでバグが発生している時間や場所を特定することができる。

5.2　プレイヤー入力の固定化

ゲームにおいてプレイヤーからの入力は制御不能なランダム要素であり、バグを再現することを困難にする。そこでプレイヤーからの入力を 1 フレーム毎に記録し、バイナリデータとして書き出す Capture モードと書き出されたバイナリデータを読み込み、プレイヤーの入力を再現する Trace モードを実装した。

6.　SPE における乱数の固定化

SPE 内で乱数を生成すると、毎回異なる値を生成してしまう。そこであらかじめ PPE 内で乱数列を生成し、inData として Task に渡しておく。Task Dandy では Task の生成、定義がされるタイミングは Super Dandy における Move 関数や Collision 関数が実行されるタイミングと同じである為、渡される乱数は Super Dandy と同じ乱数となる。(図 6)

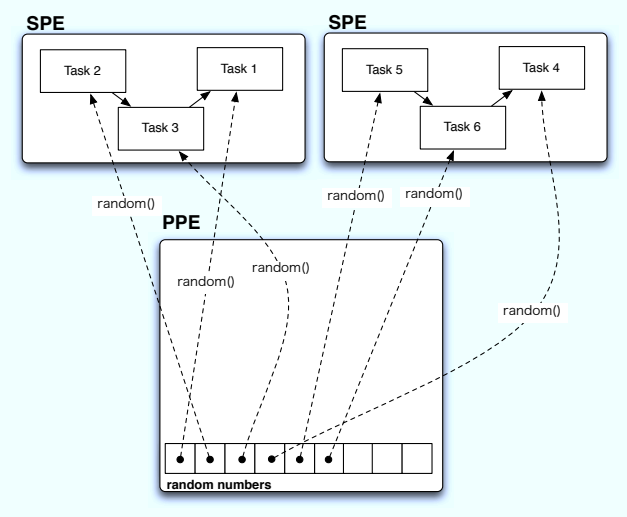

図 6　PPE 内での乱数の生成と乱数の受け渡し

6.1　描画処理を行わないビデオモード

Task Dandy ではゲームの処理や Task 生成を行った後、Rendering Engine を用いて 3 つの Task を用いて画面の描画処理を行っている。

しかし、プレイヤーの入力をバイナリデータから読み出す場合、処理の詳細が知りたい場合を除いて画面の描画処理は不要となる。そこでテスト用に画面の描画処理を行わないモードを Cerium に実装した。これは、Cerium 内で描画用の Task を生成する処理を行わないことで多くの処理時間を要する描画処理を行わずにテストを行うことができるビデオモードである。(図 7)

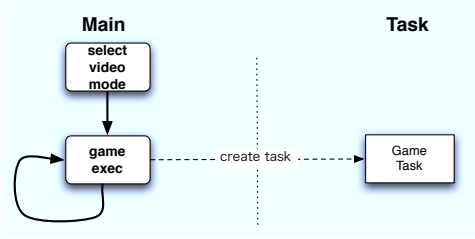

図 7　描画処理を行わないビデオモード

7.　バグ検出実験

7.1　開発した環境における検出方法

まずはじめに OpenGL バージョンの Super Dandy を Capture モードでプレイする。プレイはエンディングまで行い、プレイヤーの入力データを保存する。Task Dandy を Trace モードで実行し、入力データを読み込ませる。そして各バージョンそれぞれから得られたテストログを比較、考察し、バグの発生していると思われる箇所を特定する。

この方法で実際にテストを行い、以下のようなテストログが取れた。

```
super dandy >>
F109: DELETE [NAME]enemy_greenclab_1
[COORD]x= 56.000000  y= -24.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

F117: DELETE [NAME]enemy_greenclab_2
[COORD]x= 184.000000  y= 40.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

<< task dandy
F109: DELETE [NAME]enemy_greenclab_1
[COORD]x= 56.000000  y= -24.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

F109: DELETE [NAME]enemy_greenclab_2
[COORD]x= 184.000000  y= -24.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0
```

ここで Super Dandy と Task Dandy の 2 つのログを比較したときの特徴を洗いだすと以下のようになる。

- Super Dandy では別フレームで被弾した敵オブジェクトが Task Dandy では同フレーム内で被弾している
- 同フレーム内で被弾したときの弾丸の数が一致している
- それ以外のログは Super Dandy と Task Dandy 共に一緒である

こうした結果から 2 つのオブジェクト間で弾丸データの同期が取れていない、という仮説を立てた。Collision Task 間でデータを同期させるには、Collision Task を同じ CPU に送り、予め衝突判定に必要なパラメータの領域を LS に確保し、その領域のパラメータで衝突判定を行う方法が考えられる。

以上のことをふまえて Collision Task を書き換え、再び同じプレイヤー入力でテストログを出力させた。以下にその結果を示す。

```
super dandy>>
F109: DELETE
[NAME]enemy_greenclab_1
[COORD]x= 56.000000  y= -24.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

F117: DELETE
[NAME]enemy_greenclab_2
[COORD]x= 184.000000  y= 40.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

<< task dandy
F109: DELETE
[NAME]enemy_greenclab_1
[COORD]x= 56.000000  y= -24.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0

F117: DELETE
[NAME]enemy_greenclab_2
[COORD]x= 184.000000  y= 40.000000..
[BULLET]tlv1 = 2, tlv2 = 0 llv1 = 0
```

この結果よりオブジェクトの処理の結果がフレーム単位で同じであることがわかる。衝突判定時のテストログ出力から得られた同期バグ検出は有効であった。

8. Task への乱数受け渡しによるバグの再現性の検証

Task への乱数受け渡しの手法が期待通りの動きをするかどうか、多数の隕石オブジェクトが生成されるステージで全ての隕石オブジェクトが生成されるのを観察し、検証した。この隕石オブジェクトは以下のような実装になっている

```
int sf = random() % 4;
if ((sf == 0) || (sf == 1)) {
    p->x = -35;
    p->y = random() % (120 - 35);
    p->vx = (random() % 4 + 1);
    p->vy = random() % 3 + 1;
} else if(sf == 2) {
    ....
} else if(sf == 3) {
    ....
}
```

このオブジェクトは乱数によって 3 種類の処理に分かれる。処理の中では xy 座標、xy 方向の速度が決定し、次の状態へ遷移する、という動作になっている。そこで、この処理が行われた直後のオブジェクトの座標を記録し、Super Dandy、Task Dandy 双方のデータに違いがあるかどうか調べた。

以下はその結果である。

```
super dandy >>
...
[ID]1  [COORD]x= -35.000000  y= 20.000000
              vx= 3.000000  vy= 1.000000
...
[ID]6  [COORD]x= 220.000000  y= -30.000000
              vx= 1.000000  vy= 4.000000
...
[ID]11  [COORD]x= -35.000000  y= 57.000000
               vx= 3.000000  vy= 3.000000
...

<< task dandy
[ID]6  [COORD]x= 220.000000  y= -30.000000
              vx= 1.000000  vy= 4.000000
[ID]11  [COORD]x= -35.000000  y= 57.000000
               vx= 3.000000  vy= 3.000000
[ID]1  [COORD]x= -35.000000  y= 20.000000
              vx= 3.000000  vy= 1.000000
...
```

ID はそのオブジェクト固有の値である。SPE で並列実行させた場合、実行順序はバラバラになっているが、xy 座標、速度の値は逐次実行した場合と一致していることがわかる。Task への乱数受け渡しは有効に働いていることがわかる。

8.1 ビデオモードによる実行時間の比較

描画を行わないビデオモードと通常のビデオモードで実行時間を計測し、その差がどの程度あるのかを比較した。実行時間の計測には Unix 環境で使用できる time コマンドを使用し、計測モデルとして OpenGL で動作している Super Dandy と Task Dandy を使用した。それぞれ 1200x800 で画面描画を行うバージョンを用意し、さらに 描画を行わないビデオモードで Task Dandy を動かし、テストした。以下が計測結果である。

表 2　ビデオモードによる実行時間の比較

	OpenGL	Task	Task(no video)
実行時間	336.09 sec	6643.16 sec	385.17 sec

まず、OpenGL と Cerium の描画ありバージョンでは大きな実行時間の差が出ている。これは、Task Dandy においてゲームの Task と 描画の Task が生成されており、その処理時間が発生しているためと考えられる。また、OpenGL は GPU を用いた描画処理が可能であり、Cerium では Task を用いたソフトウェアレンダリングとなっている為、ここまでの描画時間の差が出たと思われる。

次に描画処理を行わないバージョンだが、描画処理を行った場合に比べて大幅に処理時間が短縮されている。描画処理を行わない場合、Task となっているのはゲーム内の Move と Collision だが、描画の Task に比べればそれほど大きな処理にはなっていないのがわかる。

9. ま と め

Capture と Trace によるプレイヤー入力の固定化、生成される乱数の固定化、および画面描画を行わないビデオモードの実装によってバグの再現性の向上、そしてテスト時間の短縮を実現することができた。また、テストログの出力により、Task 間の同期問題のバグを修正することも出来た。

しかし、今回のテストでは Task Dandy の数ある状態遷移の一部をテストしかされていない。さらに異なるバグを発見する為には、多種多様なゲームのプレイヤー入力を Capture、Trace する必要がある。しかし、これを人間によって行うには大きな労力を伴う。このプレイヤー入力をある程度自動的に生成できる実装が必要となる。

文　献

[1] 河野 真治, PS3 上でのゲームプログラミング, 第 51 回プログラミング・シンポジウム, 2010.

[2] 宮國　渡, Cell 用の Fine-Grain Task Manager の実装, 琉球大学理工学研究科情報工学専攻 平成 20 年度学位論文, 2009.

[3] 神里 晃, Cell を用いたゲームフレームワークの提案, 琉球大学理工学研究科情報工学専攻 平成 19 年度学位論文, 2008.

[4] 金城 裕, Fine grain Task Manager Cerium のチューニング, 日本ソフトウェア科学会第 27 回大会論文集, 2010.

[5] Mark DeLoura, Game Programming Gems, Born Digital Inc, 東京, 2002.

[6] Kent Beck, テスト駆動開発入門, テクノロジックアート, 東京, 2008.