|  |
| --- |
| "大きなゲーム"製作 プログラミング入門 |
| for XNA/C# |
| 野村 周平 |
|  |



目次

[はじめに 4](#_Toc288002858)

[この本の立ち位置・読者対象 5](#_Toc288002859)

[まずはゲームを作ってみよう 6](#_Toc288002860)

[何を作ろうか考えよう 6](#_Toc288002861)

[そのゲーム、何日で作れる？ 6](#_Toc288002862)

[大きなゲームは、小さな機能の複合体である 7](#_Toc288002863)

[仕様書を作る癖を付けよう 8](#_Toc288002864)

[360°弾避けゲームを作ってみよう 9](#_Toc288002865)

[ゲーム仕様 9](#_Toc288002866)

[まずは何も考えずに組んでみる 10](#_Toc288002867)

[完成したら動かしてみよう 17](#_Toc288002868)

[問題点を探そう 18](#_Toc288002869)

[プログラムの改良をしよう 21](#_Toc288002870)

[機能ごとに分けよう 21](#_Toc288002871)

[コメントを付ける 21](#_Toc288002872)

[サブルーチンで分割する 28](#_Toc288002873)

[マジックナンバーの定数化 36](#_Toc288002874)

[データごとにまとめよう 38](#_Toc288002875)

[あるものは使おう 39](#_Toc288002876)

[カプセル化 40](#_Toc288002877)

[オブジェクト指向の第一歩 40](#_Toc288002878)

[「動け！」だけで勝手に動くようにしよう 40](#_Toc288002879)

[ポリモーフィズム 40](#_Toc288002880)

[タスクマネージャ 40](#_Toc288002881)

[実はまだ問題がある 41](#_Toc288002882)

[メモリ管理 41](#_Toc288002883)

[もっとスマートに美しく書こう 41](#_Toc288002884)

[コーディングスタイル 41](#_Toc288002885)

[入口1つに出口1つ 41](#_Toc288002886)

[ネーミングセンス 41](#_Toc288002887)

[たとえば、弾を作る処理 41](#_Toc288002888)

[バグ対策 41](#_Toc288002889)

[テストをしよう 41](#_Toc288002890)

[手動より自動 41](#_Toc288002891)

[ユニットテストツール紹介 41](#_Toc288002892)

[あとがき 42](#_Toc288002893)

[ゲームプログラミングの心得 42](#_Toc288002894)

[できるだけ小さく作れ 42](#_Toc288002895)

[小さなゲームでも大きなゲームの作り方を 42](#_Toc288002896)

[プログラムに限った話ではない 42](#_Toc288002897)

[バシバシ公開して叩かれろ 42](#_Toc288002898)

[付録 43](#_Toc288002899)

[サンプルプログラムのダウンロード 43](#_Toc288002900)

[開発環境の設定 43](#_Toc288002901)

[索引 44](#_Toc288002902)

[筆者自己紹介 45](#_Toc288002903)

# はじめに

皆さんこんにちは。野村 周平です。この度はこの本にご興味を持って戴き、誠にありがとうございます。あなたはこの本に興味を持って戴いたということは、多少はゲーム製作に興味があることと存じます。これから実際にゲームを作りながら教えていきたいと思いますので、宜しくお願いします。

私はサラリーマンしながら副業として専門学校でゲームプログラミングの講師をやっています。講師の方は始めてからたったの一年間ですが、教える側としても色々なことがわかりました。知り合いから専門学校の学生はバカばかり、特にここ暫くはゆとりの程度が酷い、と散々なことを吹き込まれて、正直なところ自分もそういう先入観を持っていましたが、いざ蓋を開けてみると実は彼らは想像以上にレベルが高いんです。例えばクラスとは何者か一応それなりに判っているみたいですし、そこいらの入門サイトレベルの内容なら判るし組めるのです。ゲームもしょうもないワンキーゲーム程度のものでしたらちょこちょこと作れるようです。でも、その学生さんは決まってちょっと大きなゲーム（ここではマリオブラザーズは大きなゲームだと見て良いでしょう）を作らせようとすると、あっという間に破綻してしまうのです。

私はその時学生さんたちのソースコードを見て、なぜ作れないか一目で理解できました。彼らは小さなゲームの作り方しか知らないのです。それも入門サイトのサンプルコードから推察したような作り方ばかり。そこで私は書店へ行き、大きなゲームの作り方を解説している書籍を探しました。確かに幾つかの書籍は、私の教えたい内容に極めて近いものがありましたが、それらは総じて深く考えずに「ゲーム作るぞー！」と意気だっている人が興味を持てるようなものではなく、また月一度のガムとジュース代をひたすら貯め込んだ、なけなしの金で買うようなものでもありませんでした。そこで、私は筆を執り、彼ら学生さんと独学でゲームを作ろうと勉強している方々のための、大きなゲームプログラムの教科書を作ろうと決意しました。

少々前置きが長くなってしまいましたね。ここから本編に入っていきますので、皆さんどうか宜しくお願いします。

野村 周平

## この本の立ち位置・読者対象

本書は「大きなゲームプログラミング」の入門書です。（しつこいかもしれませんがマリオブラザーズは大きなゲームだと見て良いでしょう）今回非常に少ないおまじないでゲームが作れるように、とC#言語及びMicrosoft XNA Framework 3.1を選択しましたが、本書はこれらの入門書ではありません。プログラムが本当に初めてである場合は、まずC#言語の入門書とXNAの入門書をお探しの上で、本書を読むことを推奨します。下記に改めて読者対象を要約します。

* 「猫でもわかるプログラミング[[1]](#footnote-1)」など入門サイト程度のレベルのプログラムなら理解できる。
* C#とXNAの基本が解っていて、これを使った極々簡単なゲームを作ったことがある。
* でも、少しでも大きなゲームになると、あっという間に破綻してしまう。

もしあなたが上記のような方でしたら、きっとこの本は一番あなたのためになるものと思います。

# まずはゲームを作ってみよう

## 何を作ろうか考えよう

本書はゲームプログラミングの入門書ですので、早速何かゲームを作ってみましょう。あなたはどんなゲームを作るか決めていますか？もし決めていない場合、まずは何を作るか考えてみましょう。パズルでしょうか？アクションでしょうか？クイズゲームも良いでしょうし、シューティングも悪くありません。ここで一番好きなジャンルを選ぶと、後々モチベーションを維持しやすいです。ただし、RPGは少々おすすめできません。アドベンチャーも脚本と原画が既に出来上がっているとかでなければ、やめた方が良いでしょう。理由は次節で説明します。

作るゲームを決めたら、それの製作ブック専用としてノートを一冊買って、メモを取っておきましょう。この際WordやExcelを使うことはあまりお勧めできません。画面構造や簡単なラフをこれらのツールで描くのは、手書きよりもはるかに多くの時間を浪費してしまい、その間に折角のインスピレーションを逃がしてしまう場合もあります。どうしてもWordやExcelでまとめたい場合、一旦ノートに取ってから、本決まりした項目だけをそれらツールで清書していくと良いでしょう。

### そのゲーム、何日で作れる？

あなたは本書を見ているということは、少なくとも家庭用ビデオゲームを一度くらいは遊んだことありますよね？一作品でも良いので全解きしたことのある方でしたら、最後のスタッフロールを見たことがあるかと思います。普段はぼけーっと読み飛ばしてしまうかと思いますが、ここには実は製作に関わる重要な情報が隠れているのです。ゲーム雑誌とか見ると「製作期間○○カ月」とかたまに載っていますが、もし製作期間（専門用語では、「工期」と言います）とスタッフロールの人数、両方知っている場合はその二つをかけ合わせてみましょう。一つの数値が出てきましたね。これがゲームの規模「**工数**」です。

少々判りにくいかと思いますので、一つ例題を出しましょう。開発メンバーが7人いて、彼らが一晩（ここでは5時間とします）でゲームを仕上げる場合、7×5=35となります。この35が工数です。単位は「人時」と言います。人数と時間をかけているため人時です。日数でかければ人日、月数でかければ人月[[2]](#footnote-2)です。例えばこれと同じゲームを7人ではなく5人で作る場合、逆に35から5で割った数値、7時間が工期、すなわち製作時間です。1人だけでしたらもっと単純ですね。35を1で割るので、そのまま35時間かかることになります。このような計算手法を「人月計算[[3]](#footnote-3)」と言います。

人月計算の公式：

工数 = 工期 × 人数

さて、それではあなたの今考えたゲームは、どの位の製作時間を要するのでしょうか？予想してみましょう。もし初めてのゲーム製作で何カ月もかかりそうだと思う場合、それは企画を改めた方が良いでしょう。最初に作るゲームは1か月程度の工数が望ましいです。そのくらいなら途中で萎えずにモチベーションが高いまま完成を迎えられるでしょう。ゲーム製作は、あなたの想像以上に飽きやすいものです。簡単なゲームなら1か月で行けるかもしれませんが、RPGなどは1か月では少々厳しいですね。ちなみに、このようにゲーム製作にかかる工数を製作開始前に予想、あるいは計算することを「工数見積もり」と言います。これはプロのプログラマになるためには避けては通れない知識[[4]](#footnote-4)なので、できれば今のうちに覚えておいて、何か作る前に見積もりをしておくと練習になるかもしれません。

### 大きなゲームは、小さな機能の複合体である

さて、では実際に予想してみろといきなり言われても、「そんなの大きすぎて予想できないよ！」と言う方が多いのではないでしょうか？では一旦この話は置いといて、別のソフトウェアを見積もってみましょう。

あなたは「Hello, world」をご存知でしょうか？これは画面に「Hello, world!」という文字を表示するだけのソフトウェアで、プログラム言語の習得のために最初に作るものです。この程度のソフトウェアなら簡単に予想できますよね？早い人なら1分程度かもしれませんし、逆にいくら遅い人でも、恐らく1時間はかからないでしょう。

using System;

class Program

{

　static void Main(string[] args)

　{

　　Console.WriteLine("Hello, world!");

　}

}

そろそろ話を戻しましょうか。例えばあなたが、マリオブラザーズのクローンゲームを作りたいとしましょう。でも「そんなの大きすぎて予想できないよ！」となってしまった場合、予想できる範囲までゲームを分割してしまえばよいのです。

分割するためには、まずそのゲームにどんなキャラクタがいるかをリストアップして、ノートに書き込んでいきます。マリオ以外にも色々いますよね、ハエとかカメとか、あと床やPOW[[5]](#footnote-5)など当たり判定のある物体もキャラクタに含みます。次にそれぞれのキャラクタの機能をリストアップします。例えばマリオの場合、左右へ走る、ジャンプする、床やPOWを突き上げる、倒れている敵に体当たりして敵を倒す、動いている敵に体当たりして死ぬなどの機能がありますね。おっとキャラクタの描画も忘れてはいけません。この調子でほかのキャラクタもすべてリストアップしてみましょう。面倒だと投げ出す人もいるかもしれませんが、このリストは後でも使用するので、今のうちにやっておくとよいでしょう。それにゲームプログラミングはこの数倍面倒なので、この程度で面倒だと投げ出すと後々大変ですよ？さて、話を戻してキャラクタ以外の機能についても、忘れずにリストアップしましょう。例えばスコア計算や残機情報の管理及び表示、ゲームオーバー判定もありますよね？

ここまで分割できたら、そろそろ一機能単位の工数が予想できるのではないでしょうか？まだ予想が難しいようなら分割が足りない証拠でしょう。全項目の工数が予想できたら、それを全部足し合わせます[[6]](#footnote-6)。そうして出てきた数字が、そのゲームの「工数[[7]](#footnote-7)」となります。出た工数は忘れないようにノートに書いておきましょう。合計値だけではなく、機能ごとの工数もすべて書き記しておきます。そして、一つの機能を完成させたときにどのくらいの時間がかかったのか（「実績」と言います）、また実績と予想との誤差を別の色のペンで書き記します。これを専門用語で「予実管理」と言いますが、これを繰り返すうちにどの程度の誤差が出るか判るようになり、より正確な見積もりが出せるようになります。正確な見積もりが出せると、いつリリースできるかがはっきりしますし、期限に間に合うかどうかが製作開始前に判るため、機能の調整なども簡単にできます。コミックマーケット直前で間に合わなくなって、落としてしまうなんてこともなくなります！

### 仕様書を作る癖を付けよう

前回の機能分割の項目までで、おのずとそのゲームの大まかな機能一覧が見えてきたと思いますので、それをさらに事細かく書き足して、「仕様書」を作ります。また面倒だと思う人もいるかもしれませんが、仕様書がないと、集団で作るときに各個人の脳内イメージが大きくずれて大変困ります。また、一人で作るときでも仕様書があると、何を以て完成するかが把握しやすいため、面倒でも作っておきましょう。

## 360°弾避けゲームを作ってみよう

本書ではサンプル1として360°弾避けゲームを作ることにします。以下にこれを選んでみた理由を示しておきます。

1. 「小さなゲーム」でありながら「大きなゲーム」の要素を含んでいること。  
   （「大きなゲームの作り方」を知らなくても、辛うじて作れる程度である）
2. 全くプログラムを知らないのでなければ、1か月以内には作れるはずであること。
3. 後述するが、プログラムを習得したばかりの人が陥りやすいミスを再現した、問題点のあるソースコードを書きやすいこと[[8]](#footnote-8)。またその際に恐ろしく長くならないこと。

### ゲーム仕様

* シーン

SAMPLE 1

PUSH SPACE KEY.

HISCORE: 10

* + タイトル画面
    - ゲーム名は「SAMPLE 1」
    - タイトルとハイスコアを表示。
    - キーボードのスペースキーを押すとゲームシーンへ移行、ESCキーでゲーム終了。
  + ゲーム画面
    - ゲームオーバー時にタイトルシーンへ移行。
* キャラクタ

HISCORE: 10 SCORE: 10 PLAYER: \*\*

* + 主人公
    - 初期状態で画面中心にいる。
    - 大きさは64ピクセル。
    - キーボードの十字キーで上下左右に動く。画面外に移動することはできない。
    - 弾に接触するとミス。画面中の弾はクリアされる。
    - 主人公に初期状態で3回ミス猶予があり、猶予がなくなるとゲームオーバー。
  + 弾
    - 全方位の画面端の任意の位置（ランダム）から弾が出る。
    - 大きさは主人公の半分の32ピクセル。
    - ランダム等速度の自機狙い。最初は遅いが徐々に高速化。
    - 直進弾と発射後1秒ホーミングする弾の2種類。比率は8:2。
    - 直進弾は赤色、ホーミング弾は橙色。
    - 1発発射ごとにスコア+10点。
    - 最初は毎秒1発だが徐々に激化する。
* スコア
  + ゲーム開始時は0点。
  + 500点ごとにミス猶予が1回追加される。
  + 最高スコアを更新した場合、ハイスコアとしてゲーム終了時まで保持される。
* HUD
  + スコア・ハイスコア
  + ミス猶予数（残機）を星の数で表示。

### まずは何も考えずに組んでみる

そろそろ「いつになったらプログラムが組めるんだ」、と突っ込みが来そうですね。でも、もう我慢する必要はありません、皆さんお待ちかねのプログラミングの時間が来ました。XNAゲームのプロジェクトを作成し、思うがままソースコードをVisualStudioへ書き殴っていってください。

// Game.cs

using System;

using Microsoft.Xna.Framework;

using Microsoft.Xna.Framework.Graphics;

using Microsoft.Xna.Framework.Input;

namespace Sample1\_01

{

　/// <summary>

　/// This is the main type for your game

　/// </summary>

　public class Game1 : Game

　{

　　GraphicsDeviceManager graphics;

　　SpriteBatch spriteBatch;

　　Texture2D gameThumbnail;

　　SpriteFont spriteFont;

　　bool game;

　　int counter;

　　int score;

　　int prevScore;

　　int hiScore;

　　int playerAmount;

　　float playerX;

　　float playerY;

　　float[] enemyX = new float[100];

　　float[] enemyY = new float[100];

　　float[] enemySpeed = new float[100];

　　double[] enemyAngle = new double[100];

　　int[] enemyHomingAmount = new int[100];

　　bool[] enemyHoming = new bool[100];

　　public Game1()

　　{

　　　graphics = new GraphicsDeviceManager(this);

　　　Content.RootDirectory = "Content";

　　}

　　/// <summary>

　　/// LoadContent will be called once per game and is the place to load

　　/// all of your content.

　　/// </summary>

　　protected override void LoadContent()

　　{

　　　// Create a new SpriteBatch, which can be used to draw textures.

　　　spriteBatch = new SpriteBatch(GraphicsDevice);

　　　gameThumbnail = Content.Load<Texture2D>("GameThumbnail");

　　　spriteFont = Content.Load<SpriteFont>("SpriteFont");

　　}

　　/// <summary>

　　/// Allows the game to run logic such as updating the world,

　　/// checking for collisions, gathering input, and playing audio.

　　/// </summary>

　　/// <param name="gameTime">Provides a snapshot of timing values.</param>

　　protected override void Update(GameTime gameTime)

　　{

　　　KeyboardState keyState = Keyboard.GetState();

　　　if (game)

　　　{

　　　　if (keyState.IsKeyDown(Keys.Left))

　　　　{

　　　　　playerX -= 3;

　　　　}

　　　　if (keyState.IsKeyDown(Keys.Right))

　　　　{

　　　　　playerX += 3;

　　　　}

　　　　if (keyState.IsKeyDown(Keys.Up))

　　　　{

　　　　　playerY -= 3;

　　　　}

　　　　if (keyState.IsKeyDown(Keys.Down))

　　　　{

　　　　　playerY += 3;

　　　　}

　　　　if (playerX < 0)

　　　　{

　　　　　playerX = 0;

　　　　}

　　　　if (playerX > 800)

　　　　{

　　　　　playerX = 800;

　　　　}

　　　　if (playerY < 0)

　　　　{

　　　　　playerY = 0;

　　　　}

　　　　if (playerY > 600)

　　　　{

　　　　　playerY = 600;

　　　　}

　　　　if (counter % (int)MathHelper.Max(60 - counter \* 0.01f, 1) == 0)

　　　　{

　　　　　for (int i = 0; i < enemyX.Length; i++)

　　　　　{

　　　　　　if(enemyX[i] > 800 || enemyX[i] < 0 &&

　　　　　　　enemyY[i] > 600 || enemyY[i] < 0)

　　　　　　{

　　　　　　　Random rnd = new Random();

　　　　　　　int p = rnd.Next((800 + 600) \* 2);

　　　　　　　if (p < 800 || p >= 1400 && p < 2200)

　　　　　　　{

　　　　　　　　enemyX[i] = p % 800;

　　　　　　　　enemyY[i] = p < 1400 ? 0 : 600;

　　　　　　　}

　　　　　　　else

　　　　　　　{

　　　　　　　　enemyX[i] = p < 1400 ? 0 : 800;

　　　　　　　　enemyY[i] = p % 600;

　　　　　　　}

　　　　　　　enemySpeed[i] = rnd.Next(1, 3) + counter \* 0.001f;

　　　　　　　enemyAngle[i] = Math.Atan2(

　　　　　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　　　　　　enemyHoming[i] = rnd.Next(100) >= 80;

　　　　　　　enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;

　　　　　　　score += 10;

　　　　　　　if (score % 500 < prevScore % 500)

　　　　　　　{

　　　　　　　　playerAmount++;

　　　　　　　}

　　　　　　　prevScore = score;

　　　　　　　if (hiScore < score)

　　　　　　　{

　　　　　　　　hiScore = score;

　　　　　　　}

　　　　　　　break;

　　　　　　}

　　　　　}

　　　　}

　　　　bool hit = false;

　　　　for (int i = 0; i < enemyX.Length; i++)

　　　　{

　　　　　if (Math.Abs(playerX - enemyX[i]) < 48 &&

　　　　　　Math.Abs(playerY - enemyY[i]) < 48)

　　　　　{

　　　　　　hit = true;

　　　　　　game = --playerAmount >= 0;

　　　　　　break;

　　　　　}

　　　　　if (--enemyHomingAmount[i] > 0)

　　　　　{

　　　　　　enemyAngle[i] = Math.Atan2(

　　　　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　　　　}

　　　　　enemyX[i] += (float)Math.Cos(enemyAngle[i]) \* enemySpeed[i];

　　　　　enemyY[i] += (float)Math.Sin(enemyAngle[i]) \* enemySpeed[i];

　　　　}

　　　　if (hit)

　　　　{

　　　　　for (int i = 0; i < enemyX.Length; i++)

　　　　　{

　　　　　　enemyX[i] = -32;

　　　　　　enemyY[i] = -32;

　　　　　　enemySpeed[i] = 0;

　　　　　}

　　　　}

　　　　counter++;

　　　}

　　　else

　　　{

　　　　if (keyState.IsKeyDown(Keys.Escape))

　　　　{

　　　　　Exit();

　　　　}

　　　　if (keyState.IsKeyDown(Keys.Space))

　　　　{

　　　　　game = true;

　　　　　playerX = 400;

　　　　　playerY = 300;

　　　　　counter = 0;

　　　　　score = 0;

　　　　　prevScore = 0;

　　　　　playerAmount = 2;

　　　　　for (int i = 0; i < enemyX.Length; i++)

　　　　　{

　　　　　　enemyX[i] = -32;

　　　　　　enemyY[i] = -32;

　　　　　　enemySpeed[i] = 0;

　　　　　}

　　　　}

　　　}

　　　base.Update(gameTime);

　　}

　　/// <summary>

　　/// This is called when the game should draw itself.

　　/// </summary>

　　/// <param name="gameTime">Provides a snapshot of timing values.</param>

　　protected override void Draw(GameTime gameTime)

　　{

　　　GraphicsDevice.Clear(Color.CornflowerBlue);

　　　spriteBatch.Begin();

　　　if (game)

　　　{

　　　　spriteBatch.Draw(

　　　　　gameThumbnail, new Vector2(playerX, playerY), null,

　　　　　Color.White, 0f, new Vector2(32, 32), 1f,

　　　　　SpriteEffects.None, 0f);

　　　　for (int i = 0; i < enemyX.Length; i++)

　　　　{

　　　　　spriteBatch.Draw(

　　　　　　gameThumbnail, new Vector2(enemyX[i], enemyY[i]), null,

　　　　　　enemyHoming[i] ? Color.Orange : Color.Red, 0f,

　　　　　　new Vector2(32, 32), 0.5f, SpriteEffects.None, 0f);

　　　　}

　　　　spriteBatch.DrawString(spriteFont, "SCORE: " + score.ToString(),

　　　　　new Vector2(300, 560), Color.Black);

　　　　spriteBatch.DrawString(spriteFont,

　　　　　"PLAYER: " + new string('\*', playerAmount),

　　　　　new Vector2(600, 560), Color.Black);

　　　}

　　　else

　　　{

　　　　spriteBatch.DrawString(spriteFont, "SAMPLE 1", new

　　　　　Vector2(200, 100), Color.Black, 0f, Vector2.Zero, 5f,

　　　　　SpriteEffects.None, 0f);

　　　　spriteBatch.DrawString(spriteFont, "PUSH SPACE KEY.",

　　　　　new Vector2(340, 400), Color.Black);

　　　}

　　　spriteBatch.DrawString(spriteFont, "HISCORE: " + hiScore.ToString(),

　　　　new Vector2(0, 560), Color.Black);

　　　spriteBatch.End();

　　　base.Draw(gameTime);

　　}

　}

}

上記コードはすべてXNAプロジェクト作成時に自動的に作成されるGame1.csをベースに書き足しています。（Sample1\_01）また、コンテンツとしてGameThumbnail（サムネイル画像をコピー）とSpriteFont（初期状態）を追加しています。

### 完成したら動かしてみよう

さて完成したので、早速ですが動かしてみましょう。



どうやら、ちゃんと動いているようですね。仕様通り自機は十字キーで動き、赤弾は直進、橙弾はホーミングで自機を目指して飛んできます。残機が500点ごとに増えていき、逆に残機がない状態でミスをするとゲームオーバーとなり、タイトル画面に戻ります。ハイスコアはソフトウェア終了時までちゃんと保持されますね。

では、ここまでできたところで、早速このゲームをさらに改造してやりこみ要素や追加ステージ、はたまた協力プレー機能──などを組み込んでいくと、**あっという間にソースコードが破綻して**しまいます。その前にするべきことがあるのです。それを確認するために、次章から書いたソースコードを確認していきましょう。



# 問題点を探そう

前章で一つゲームのソースコードを書きましたが、ここではそれの問題点を探して、修正してみましょう。ここから先を読む前に、皆さんも何がまずいのか、予め予想してみて、メモしましょう。

前章のソースコードはいくつかの問題点を孕んでいますが、中でも最も大きいのが、「**小さいゲーム向けのソースコードの組み方である**」ことです。大きなゲームを作るためには、大きなゲームに適したプログラミングをすることが必要です。小さいゲーム向けのプログラミング手法のままで大きなゲームを書くと、大抵途中で破綻します[[9]](#footnote-9)。では、大きなゲーム向けのソースコードとは一体どのようなものなのでしょうか？

実際には、そんな一言で言えるほど簡単な正解ではありませんが、誤解を恐れず極論を言うと、「**小さいソースコードを書くこと**」が正解です。「なぜ大きなゲームを作るのに小さいソースコードを書かなきゃいけないんだ？どう考えても逆じゃないか？これは禅問答か？」と、疑問に思う方も多いのではないかと思いますが、謎かけでも頓智でも何でもなく、大きいゲームを作るためには小さいソースコードを書いて、その小さいソースコード同士を繋げていって大きくしていくのが、破綻せずに完成させる最大の近道なのです。

それでは、大きいソースコードと小さいソースコードの違いとは何なのでしょうか？正解を挙げたところで、前章のソースコードをもう一度見てみましょう。そして、最初に挙げた「Hello, world」も見て、両者のソースコードの違いをよく考えてみましょう。「Hello, world」の方が圧倒的に見やすいことに、恐らく皆さんは気付くはずです。そうです、小さいソースコードを書くと、その分見やすくなるのです。実際に前章のソースコードにある、各メソッドの行数を数えると、下記のようになります。

* Game1コンストラクタ 2行
* LoadContentメソッド 4行
* Updateメソッド 125行
* Drawメソッド 34行

Game1コンストラクタとLoadContentメソッドは良いとしても、Updateメソッドは多すぎですね。1メソッドだけで125行と、相当な行数を消費しています。Drawメソッドもやや膨れていますね。このゲームを選定した理由として、コードが長くなり過ぎないことと言いましたが、それでも何も考えずに書くと、ここまで膨れ上がってしまいました。私個人的には1メソッド内のコード量は多くても30行～40行、できるなら20行弱に抑えるべきだと思います。それ以上増えると、ソースコードの可読性が大きく落ちます。可読性が悪いと改修するのも大変です。下手すると右を叩けば左が出て、左を叩けば右が出て、と目も当てられない状況に陥ることもあります。ちなみに、そのような可読性の低下を補うために、コメントやドキュメンテーション[[10]](#footnote-10)がありますが、これに頼りすぎてコードの短縮化を怠ることは、正直な話推奨しません。

少々私の経験談となりますが、私がサラリーマンとしてプログラマを始めたばかりの頃、ある現場に派遣されて、そこでWebアプリケーションの改修作業に従事させられました。その時使用していた言語はC#ではなくJava1.3でしたが、1メソッド辺り数千行はあるソースコードが平気で出てくるのです。そしてメソッド内のコードも、お世辞にも洗練されているとは言えず、私は一目見て、そのコードを読むのをやめました。そのメソッドのドキュメンテーション、即ち上っ面だけを見て、そういう機能なのだろうと思ってそのメソッドを利用していたのですが、どうも想定通りの動作をしてくれないのです。最初自分のところのコードの書き方、あるいは呼び出し方がおかしいのかと思っていたのですが、そのメソッドの代わりにスタブを呼んでみたりしたところ、どうやらそうでもないのです。まさかなぁ……と思いそのメソッドを作った人に事情を聴いたら、なんと今は内部構造が変わって全く別のメソッドになっていたとのことです。つまりそのコメントは良く言えば古いバージョン用の解説、悪く言えば全くの嘘っぱちだったのです。このときはすぐ近くに当事者がいたため助かりましたが、もしそうでもなかったことを考えると、ぞっとします。

ここまで読んで、「コメントとソースコードの同期を、徹底化すればいいじゃないか」と思う方が恐らく多いのではないでしょうか。事実それが正解です。しかし実際のところ、ドキュメンテーション、あるいはコメントとソースコードを同期させ続けるのは、かなりの体力と時間の浪費が必要です。私はそうまでして無理してコメントを同期させるより、コメントはシンプルに書き（それこそ「○○します」の一言程度で十分）、余った作業時間を内部ソースコードの短縮化・洗練化に充てた方が有意義だと思います。あなたがゲームライブラリやエンジンを作っているわけでなければ、大量のコメントとドキュメンテーションで説明されるより、とても短くてユーザが3秒そのメソッドの中身を見れば、その機能がなんであるかを理解できるのが私にとっての理想です[[11]](#footnote-11)。

そう言うと稀に、行数を縮めるために横に伸ばす人も見かけますが、これは一層可読性を落としてしまい本末転倒です。桁数も100～120桁程度[[12]](#footnote-12)に抑えるのが望ましいでしょう。

どうでもいい話ですが、私が鼻水垂らして小学校通っていた頃は、ソースコードの単価は印刷した紙のグラム単位で、金額を水増しするために、ループなどをすべてインライン展開してコード量を増やしていたそうです。そんな話を年配の方々からよく聞きますが、私はきっと都市伝説だろう、と信じています。

それでは少々話がそれてしまいましたが、次のページからは「では小さいソースコードにするためにはどのようにすればよいのか」を解説していきたいと思います。

# プログラムの改良をしよう

本章では、サンプル1「360°弾避けゲーム」について、前章で挙げた問題点を修正していきます。とは言っても、いきなり大きく改変したりすることはありません。そんなことしても、きっと皆さんは混乱してしまうでしょう[[13]](#footnote-13)。みなさんが理解しやすいよう、少しずつ改変していきます。

## 機能ごとに分けよう

前章では「大きなゲームを作るためには、小さいソースコードを書く」と述べましたね。では、ゲームの大きさを維持したままソースコードを小さくするにはどうすればよいでしょう[[14]](#footnote-14)？何も、ソースコード全体を縮める必要はありません。ゲームを機能ごとに分割すれば、小さく見えますよね？

### コメントを付ける

私は前章でも述べたとおり、あまり几帳面にコメントを書きすぎることは、あまり望ましくないと考えています。しかし、自分用の一時的なメモ書き程度なら、積極的に活用するのも問題ないでしょう。一方ドキュメンテーションはprivateなメソッドやフィールドであっても、一応書いておくべきです。但し、やはり内容はゲームライブラリ（ゲームエンジン）などを作っているわけでない場合「○○をします」程度で十分でしょう。

下記に示したコードでは、前回のサンプルから変更を加えていない部分については省略しています。（Sample1\_02）

// フィールド宣言

　/// <summary>グラフィック デバイス構成管理。</summary>

　GraphicsDeviceManager graphics;

　/// <summary>スプライト バッチ。</summary>

　SpriteBatch spriteBatch;

　/// <summary>キャラクタ用画像。</summary>

　Texture2D gameThumbnail;

　/// <summary>フォント画像。</summary>

　SpriteFont spriteFont;

　/// <summary>ゲーム中かどうか。</summary>

　bool game;

　/// <summary>ゲームの進行カウンタ。</summary>

　int counter;

　/// <summary>現在のスコア。</summary>

　int score;

　/// <summary>前フレームのスコア。</summary>

　int prevScore;

　/// <summary>ハイスコア。</summary>

　int hiScore;

　/// <summary>ミス猶予(残機)数。</summary>

　int playerAmount;

　/// <summary>プレイヤーのX座標。</summary>

　float playerX;

　/// <summary>プレイヤーのY座標。</summary>

　float playerY;

　/// <summary>敵のX座標一覧。</summary>

　float[] enemyX = new float[100];

　/// <summary>敵のY座標一覧。</summary>

　float[] enemyY = new float[100];

// Update()メソッド内部

KeyboardState keyState = Keyboard.GetState();

if (game)

{

　// ゲーム画面

　// プレイヤー移動処理

　if (keyState.IsKeyDown(Keys.Left))

　{

　　playerX -= 3;

　}

　if (keyState.IsKeyDown(Keys.Right))

　{

　　playerX += 3;

　}

　if (keyState.IsKeyDown(Keys.Up))

　{

　　playerY -= 3;

　}

　if (keyState.IsKeyDown(Keys.Down))

　{

　　playerY += 3;

　}

　/// <summary>敵の移動速度一覧。</summary>

　float[] enemySpeed = new float[100];

　/// <summary>敵の移動角度一覧。</summary>

　double[] enemyAngle = new double[100];

　/// <summary>敵のホーミング有効時間。</summary>

　int[] enemyHomingAmount = new int[100];

　/// <summary>ホーミング対応の敵かどうか。</summary>

　bool[] enemyHoming = new bool[100];

　if (playerX < 0)

　{

　　playerX = 0;

　}

　if (playerX > 800)

　{

　　playerX = 800;

　}

　if (playerY < 0)

　{

　　playerY = 0;

　}

　if (playerY > 600)

　{

　　playerY = 600;

　}

　// 弾の生成

　if (counter % (int)MathHelper.Max(60 - counter \* 0.01f, 1) == 0)

　{

　　for (int i = 0; i < enemyX.Length; i++)

　　{

　　　if (enemyX[i] > 800 || enemyX[i] < 0 &&

　　　　enemyY[i] > 600 || enemyY[i] < 0)

　　　{

　　　　Random rnd = new Random();

　　　　int p = rnd.Next((800 + 600) \* 2);

　　　　if (p < 800 || p >= 1400 && p < 2200)

　　　　{

　　　　　enemyX[i] = p % 800;

　　　　　enemyY[i] = p < 1400 ? 0 : 600;

　　　　}

　　　　else

　　　　{

　　　　　enemyX[i] = p < 1400 ? 0 : 800;

　　　　　enemyY[i] = p % 600;

　　　　}

　　　　enemySpeed[i] = rnd.Next(1, 3) + counter \* 0.001f;

　　　　enemyAngle[i] = Math.Atan2(

　　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　　　enemyHoming[i] = rnd.Next(100) >= 80;

　　　　enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;

　　　　score += 10;

　　　　if (score % 500 < prevScore % 500)

　　　　{

　　　　　playerAmount++;

　　　　}

　　　　prevScore = score;

　　　　if (hiScore < score)

　　　　{

　　　　　hiScore = score;

　　　　}

　　　　break;

　　　}

　　}

　}

　// 弾の移動、及び接触判定

　bool hit = false;

　for (int i = 0; i < enemyX.Length; i++)

　{

　　if (Math.Abs(playerX - enemyX[i]) < 48 &&

　　　Math.Abs(playerY - enemyY[i]) < 48)

　　{

　　　hit = true;

　　　game = --playerAmount >= 0;

　　　break;

　　}

　　if (--enemyHomingAmount[i] > 0)

　　{

　　　enemyAngle[i] = Math.Atan2(

　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　}

　　enemyX[i] += (float)Math.Cos(enemyAngle[i]) \* enemySpeed[i];

　　enemyY[i] += (float)Math.Sin(enemyAngle[i]) \* enemySpeed[i];

　}

　if (hit)

　{

　　for (int i = 0; i < enemyX.Length; i++)

　　{

　　　enemyX[i] = -32;

　　　enemyY[i] = -32;

　　　enemySpeed[i] = 0;

　　}

　}

　counter++;

}

else

{

　// タイトル画面

　if (keyState.IsKeyDown(Keys.Escape))

　{

　　Exit();

　}

　if (keyState.IsKeyDown(Keys.Space))

　{

　　// ゲーム開始

　　game = true;

　　playerX = 400;

　　playerY = 300;

　　counter = 0;

　　score = 0;

　　prevScore = 0;

　　playerAmount = 2;

// Draw()メソッド内

GraphicsDevice.Clear(Color.CornflowerBlue);

spriteBatch.Begin();

if (game)

{

　// ゲーム描画

　// 自機の描画

　spriteBatch.Draw(

　　gameThumbnail, new Vector2(playerX, playerY), null,

　　Color.White, 0f, new Vector2(32, 32), 1f,

　　SpriteEffects.None, 0f);

　// 敵機の描画

　for (int i = 0; i < enemyX.Length; i++)

　{

　　spriteBatch.Draw(

　　　gameThumbnail, new Vector2(enemyX[i], enemyY[i]), null,

　　　enemyHoming[i] ? Color.Orange : Color.Red, 0f,

　　　new Vector2(32, 32), 0.5f, SpriteEffects.None, 0f);

　}

　// HUDの描画

　spriteBatch.DrawString(spriteFont, "SCORE: " + score.ToString(),

　　new Vector2(300, 560), Color.Black);

　spriteBatch.DrawString(spriteFont,

　　"PLAYER: " + new string('\*', playerAmount),

　　for (int i = 0; i < enemyX.Length; i++)

　　{

　　　enemyX[i] = -32;

　　　enemyY[i] = -32;

　　　enemySpeed[i] = 0;

　　}

　}

}

base.Update(gameTime);

　　new Vector2(600, 560), Color.Black);

}

else

{

　// タイトル画面

　spriteBatch.DrawString(spriteFont, "SAMPLE 1", new

　　Vector2(200, 100), Color.Black, 0f, Vector2.Zero, 5f,

　　SpriteEffects.None, 0f);

　spriteBatch.DrawString(spriteFont, "PUSH SPACE KEY.",

　　new Vector2(340, 400), Color.Black);

}

spriteBatch.DrawString(spriteFont, "HISCORE: " + hiScore.ToString(),

　new Vector2(0, 560), Color.Black);

spriteBatch.End();

base.Draw(gameTime);

Game1クラスの全フィールドにドキュメンテーションを追加し、またUpdateメソッドやDrawメソッドの大まかな機能の区切りの部分にコメントを加えました。

### サブルーチンで分割する

サブルーチンとは、端的に述べるとプログラム上における、一定の機能の集合体のことです。サブルーチンというキーワードは、最近では耳にしないですね。今風にいうとメソッドとか言います。両者は同じように見えて微妙に違うのですが、この項ではメソッドをあたかもサブルーチンのように使うためこのように呼んでいます。これは、特に難しいことではありません。すでにUpdateメソッドとDrawメソッドに分かれているので、それをさらに細分化してやるだけです。

さて、前回機能ごとの大まかな区切りでコメントを付けましたので、今度はそれを目印にサブルーチンで分割していきます（Sample1\_03）。

/// <summary>

/// Allows the game to run logic such as updating the world,

/// checking for collisions, gathering input, and playing audio.

/// </summary>

/// <param name="gameTime">Provides a snapshot of timing values.</param>

protected override void Update(GameTime gameTime)

{

　KeyboardState keyState = Keyboard.GetState();

　if (game)

　{

　　movePlayer(keyState);

　　createEnemy();

　　if (enemyMoveAndHitTest())

　　{

　　　enemyReset();

　　}

　　counter++;

　}

　else

　{

　　updateTitle(keyState);

　}

　base.Update(gameTime);

}

/// <summary>

/// 自機を移動します。

/// </summary>

/// <param name="keyState">現在のキー入力状態。</param>

private void movePlayer(KeyboardState keyState)

{

　if (keyState.IsKeyDown(Keys.Left))

　{

　　playerX -= 3;

　}

　if (keyState.IsKeyDown(Keys.Right))

　{

　　playerX += 3;

　}

　if (keyState.IsKeyDown(Keys.Up))

　{

　　playerY -= 3;

　}

　if (keyState.IsKeyDown(Keys.Down))

　{

　　playerY += 3;

　}

　if (playerX < 0)

　{

　　playerX = 0;

　}

　if (playerX > 800)

　{

　　playerX = 800;

　}

　if (playerY < 0)

　{

　　playerY = 0;

　}

　if (playerY > 600)

　{

　　playerY = 600;

　}

}

/// <summary>

/// 敵機を作成します。

/// </summary>

private void createEnemy()

{

　if (counter % (int)MathHelper.Max(60 - counter \* 0.01f, 1) == 0)

　{

　　for (int i = 0; i < enemyX.Length; i++)

　　{

　　　if (enemyX[i] > 800 || enemyX[i] < 0 &&

　　　　enemyY[i] > 600 || enemyY[i] < 0)

　　　{

　　　　Random rnd = new Random();

　　　　int p = rnd.Next((800 + 600) \* 2);

　　　　if (p < 800 || p >= 1400 && p < 2200)

　　　　{

　　　　　enemyX[i] = p % 800;

　　　　　enemyY[i] = p < 1400 ? 0 : 600;

　　　　}

　　　　else

　　　　{

　　　　　enemyX[i] = p < 1400 ? 0 : 800;

　　　　　enemyY[i] = p % 600;

　　　　}

　　　　enemySpeed[i] = rnd.Next(1, 3) + counter \* 0.001f;

　　　　enemyAngle[i] = Math.Atan2(

　　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　　　enemyHoming[i] = rnd.Next(100) >= 80;

　　　　enemyHomingAmount[i] = enemyHoming[i] ? 60 : 0;

　　　　score += 10;

　　　　if (score % 500 < prevScore % 500)

　　　　{

　　　　　playerAmount++;

　　　　}

　　　　prevScore = score;

　　　　if (hiScore < score)

　　　　{

　　　　　hiScore = score;

　　　　}

　　　　break;

　　　}

　　}

　}

}

/// <summary>

/// 敵機の移動、及び接触判定をします。

/// </summary>

/// <returns>接触した場合、true。</returns>

private bool enemyMoveAndHitTest()

{

　bool hit = false;

　for (int i = 0; i < enemyX.Length; i++)

　{

　　if (Math.Abs(playerX - enemyX[i]) < 48 &&

　　　Math.Abs(playerY - enemyY[i]) < 48)

　　{

　　　hit = true;

　　　game = --playerAmount >= 0;

　　　break;

　　}

　　if (--enemyHomingAmount[i] > 0)

　　{

　　　enemyAngle[i] = Math.Atan2(

　　　　playerY - enemyY[i], playerX - enemyX[i]);

　　}

　　enemyX[i] += (float)Math.Cos(enemyAngle[i]) \* enemySpeed[i];

　　enemyY[i] += (float)Math.Sin(enemyAngle[i]) \* enemySpeed[i];

　}

　return hit;

}

/// <summary>

/// 敵機を初期状態にリセットします。

/// </summary>

private void enemyReset()

{

　for (int i = 0; i < enemyX.Length; i++)

　{

　　enemyX[i] = -32;

　　enemyY[i] = -32;

　　enemySpeed[i] = 0;

　}

}

/// <summary>

/// タイトル画面を更新します。

/// </summary>

/// <param name="keyState">現在のキー入力状態。</param>

private void updateTitle(KeyboardState keyState)

{

　if (keyState.IsKeyDown(Keys.Escape))

　{

　　Exit();

　}

　if (keyState.IsKeyDown(Keys.Space))

　{

　　// ゲーム開始

　　game = true;

　　playerX = 400;

　　playerY = 300;

　　counter = 0;

　　score = 0;

　　prevScore = 0;

　　playerAmount = 2;

　　for (int i = 0; i < enemyX.Length; i++)

　　{

　　　enemyX[i] = -32;

　　　enemyY[i] = -32;

　　　enemySpeed[i] = 0;

　　}

　}

}

/// <summary>

/// This is called when the game should draw itself.

/// </summary>

/// <param name="gameTime">Provides a snapshot of timing values.</param>

protected override void Draw(GameTime gameTime)

{

　GraphicsDevice.Clear(Color.CornflowerBlue);

　spriteBatch.Begin();

　if (game)

　{

　　drawGame();

　}

　else

　{

　　drawTitle();

　}

　spriteBatch.DrawString(spriteFont, "HISCORE: " + hiScore.ToString(),

　　new Vector2(0, 560), Color.Black);

　spriteBatch.End();

　base.Draw(gameTime);

}

/// <summary>

/// タイトル画面を描画します。

/// </summary>

private void drawTitle()

{

　spriteBatch.DrawString(spriteFont, "SAMPLE 1", new

　　Vector2(200, 100), Color.Black, 0f, Vector2.Zero, 5f,

　　SpriteEffects.None, 0f);

　spriteBatch.DrawString(spriteFont, "PUSH SPACE KEY.",

　　new Vector2(340, 400), Color.Black);

}

/// <summary>

/// ゲーム画面を描画します。

/// </summary>

private void drawGame()

{

　drawPlayer();

　drawEnemy();

　drawHUD();

}

/// <summary>

/// 自機を描画します。

/// </summary>

private void drawPlayer()

{

　spriteBatch.Draw(

　　gameThumbnail, new Vector2(playerX, playerY), null,

　　Color.White, 0f, new Vector2(32, 32), 1f,

　　SpriteEffects.None, 0f);

}

/// <summary>

/// 敵機を描画します。

/// </summary>

private void drawEnemy()

{

　for (int i = 0; i < enemyX.Length; i++)

　{

　　spriteBatch.Draw(

　　　gameThumbnail, new Vector2(enemyX[i], enemyY[i]), null,

　　　enemyHoming[i] ? Color.Orange : Color.Red, 0f,

　　　new Vector2(32, 32), 0.5f, SpriteEffects.None, 0f);

　}

}

/// <summary>

/// HUDを描画します。

/// </summary>

private void drawHUD()

{

　spriteBatch.DrawString(spriteFont, "SCORE: " + score.ToString(),

　　new Vector2(300, 560), Color.Black);

　spriteBatch.DrawString(spriteFont,

　　"PLAYER: " + new string('\*', playerAmount),

　　new Vector2(600, 560), Color.Black);

}

これだけでもUpdateメソッドとDrawメソッドは相当にすっきりしてきましたね。次節からこのコードをさらに整理していきます。

### マジックナンバーの定数化

XNAはプロジェクトを自動生成すると、初期設定では画面サイズがSVGA(800×600)となっています。今回これに合わせて作っていきましたが、もし例えばここで「やっぱVGA(640×480)の方が昔からのゲームの標準だから、そっちに合わせて」などと言われたら、あなたならどのように対応しますか？このゲームの場合、座標を決め打ちで入れている個所は、どう少なく見積もっても10箇所は上回ります。1個1個手入力で直していきますか？もし仮にそれで対応できたとしても、後になって「やっぱXBOX360でワイド表示に対応させるために720p(1280×720)にして」とか言われたら、まさに発狂ものでしょう。今の話は極端な例えでしたが、実際にそうでなくても一つ一つ手入力での修正は、修正漏れや誤入力などのミスの大きな原因となりえます。また、このように何の脈絡もなく現れる数値のことを「マジックナンバー」と呼び、可読性の低下を招きます。

このようなマジックナンバーには、人間に理解しやすい別名を与えるべきです。これを一般的に「定数化」と言います。例えばいきなり「800」と言う数値が出てきても、そのコードの作者でない誰かが見た場合、もしかしたら前後の文脈から判断して、画面サイズの横幅と気付く人もいるかもしれませんが、あなたがトリッキーな呪文を書くことを好む場合、前後の文脈から判断しきれず、それは謎の数値として見做されてしまうでしょう。そこで「CREEN\_WIDTH」などと別名を与えてやると、そこだけ見てこれは画面の横幅か、と誰が見ても即座に認識できるようになります。

下記のサンプル（Sample1\_04）では、マジックナンバーをconst定数、またはreadonly変数で定義していますが、場合によっては列挙体を使用したり、Singletonパターン[[15]](#footnote-15)を応用した静的なインスタンスを使ったりした方がスマートな場合もあるでしょう。また、その箇所でしか使用しないことが明白なマジックナンバーは、定数をローカルスコープで定義する方が良いでしょう。

TODO : コード挿入（1\_04マジックナンバー定数化）

フル

## データごとにまとめよう

このゲームでは最大100個の弾が自機目がけて襲い掛かってきます。100個の弾を同時に動かすためには、100個分の位置、100個分の速度、100個分の角度が必要となります。さらにこのゲームではホーミング弾かどうかと、ホーミングの残り持続時間もそれぞれ100個分必要となりますね。ところで、これらのデータにアクセスするためには、どのようにやっていたか思い出してください。例えばここでは58番目の弾のデータを取り出すと仮定しましょう。日本語で示すと大体下記のような感じだったはずです。

1. X座標一覧のうち58番目の値を取り出す。
2. Y座標一覧のうち58番目の値を取り出す。
3. 速度一覧のうち58番目の値を取り出す。
4. 角度一覧のうち58番目の値を取り出す。
5. ホーミングフラグ一覧のうち58番目の値を取り出す。
6. ホーミング持続時間残数一覧のうち58番目の値を取り出す。

これでもまぁ、悪くはないと思いますが、下記のやり方の方がスマートに見えませんか？

1. 弾情報ファイル一覧のうち、58番目の弾情報データを取り出す。
2. 1で取り出したデータからX座標を取り出す。
3. 1で取り出したデータからY座標を取り出す。
4. 1で取り出したデータから速度を取り出す。
5. 1で取り出したデータから角度を取り出す。
6. 1で取り出したデータからホーミングフラグを取り出す。
7. 1で取り出したデータからホーミング持続時間残数を取り出す。

一つのデータに複数の値を入れるにはどうすればよいでしょうか？プログラミングの基本を勉強してきたあなたなら判るでしょう。構造体を使えばよいのです。

TODO : コード挿入（1\_05構造体導入）

フル

構造体の宣言はGame1.csとは別ファイルにするとよいでしょう。そのクラスでしか使わず、外部に公開したくないなど、よほど強い理由がない限りは極力1ファイル1クラス、または1構造体にすることを心がけましょう。もしフォルダを刻みたい場合、名前空間に合わせるとよいでしょう。

さて、今までゲームクラスでは弾のX座標・Y座標・速度・角度などを個別に管理していましたが、このサンプル（Sample1\_05）では弾情報の構造体のみを管理すればよいようになりました。

### あるものは使おう

「車輪の再発明」と言う慣用句をご存知ですか？その名の通り、すでに存在しているものをもう一度作ってしまうことを指します。例えばパソコンで動作するOSと聞くと何を思い浮かべますか？読者の皆さんなら、恐らく真っ先にWindowsが出てくるかと思います。他にも挙げてみましょう。Macの標準OS「OS X」があります。他にもLinuxやBSDなどUNIXベースのOSもあります（OS XもBSDの仲間です）。もうちょっとマイナーどころとなると、BeOSやMonaOSなんかもありますね。車輪の再発明とは、これらのOSが信用できない、あるいはほかの理由で自家製OSを作ってしまうことです。これがもっとひどくなると、社内では自家製・または自社製ソフトウェアしか価値を認めず、存在を認めず、当然運用も認めない[[16]](#footnote-16)などとなり、「自前主義」とか「NIH症候群」などと呼ばれるようになります。

車輪の再発明は、学習・研究目的で行う場合は、ブラックボックスの内部構造が理解できるなど、あなたにとってメリットとなることが多いでしょう。但し、それ以外の目的で行うことはデメリットの方が大きく、お勧めできません。使いたいソフトウェアの品質が目に余るほど低く、それの代替版を作るなどの明確な目的がある場合を除き、再発明版の方がクオリティの低いソフトウェアとなってしまう場合が殆どです[[17]](#footnote-17)。私は、そのような車輪の再発明に労力を割くくらいなら、類似しているソフトウェアを探し、仕様を把握してそれに合わせた構造を作る方が、有意義な時間を使えると考えます。

少々脱線しかけたので話を戻しましょう。前項では弾の情報を管理するために、構造体を作りましたね。そこではX座標とY座標、速度と角度、そしてホーミング制御用のデータが2つの合計6つのデータがありました。しかしXNAに標準で入っているVector2型を使用すると、これが4つのデータで済みます。Vector2型にはXとYの二つの変数があり、それを使い位置情報を保持できるだけでなく、速度や方角を保持するためにも活用することができます。型名を見る限り、むしろこちらの使い方が本来のものであることに気づきますね。

下記のソースコード（Sample1\_06）では、自機と弾の座標と速度角度管理にVector2型を使用しています。固定長配列の代わりに可変長配列を使っているところや、速度角度計算に三角関数を使わず、ベクトル計算と四元数で代用しているところなども、よく比較してみてください。

TODO : コード挿入（1\_06あるものは使いまわす）

各関数内のみ

### カプセル化

前々回で座標や速度などの弾情報を構造体という形で1つのデータとしてまとめました。ところで構造体にはどのようなものが入れられるのでしょうか？列挙してみましょう。

* 変数
* 関数（メソッド）
* イベント

構造体には上記のように変数以外のもの、例えば関数なども入れることができます。そうすると、前章で分割したサブルーチンの一部は、構造体の中に埋め込んでしまった方が自然に見えますよね？さっそくコードを書き直してみましょう（Sample1\_07）。このように構造体の中に埋め込むことは、他のメリット（カプセル化やポリモーフィズムなど）もあるのですが、それは後述します。

TODO : コード挿入（1\_07メソッド導入・弾作成はstaticで）

フル

弾作成処理のみstaticメソッドになっていますが、呼び出し方の違いや、ほかのメソッドとの違い、そもそもstaticメソッドにした理由を、各自で考えてみましょう。

### オブジェクト指向の第一歩

## 「動け！」だけで勝手に動くようにしよう

### ポリモーフィズム

### タスクマネージャ

GameComponentをタスクに使用しない理由とか

## 実はまだ問題がある

### メモリ管理

XBOX360で動かすと遅い

#### 解決策

##### Flyweightパターン

##### Stateパターン

###### 継承よりも移譲

###### 分岐よりも移譲

Singletonパターン

## もっとスマートに美しく書こう

### コーディングスタイル

### 入口1つに出口1つ

### ネーミングセンス

### たとえば、弾を作る処理

コピペはバグの温床

#### Builderパターン

#### せっかちな人はFaçadeパターン

## バグ対策

### テストをしよう

### 手動より自動

### ユニットテストツール紹介

# あとがき

## ゲームプログラミングの心得

### できるだけ小さく作れ

### 小さなゲームでも大きなゲームの作り方を

ただし学習中に限る

### プログラムに限った話ではない

### バシバシ公開して叩かれろ

ソースも公開してしまえ

# 付録

## サンプルプログラムのダウンロード

## 開発環境の設定

## 索引

Hello, world, 7

Singletonパターン, 37

工期, 6

工数, 6

工数見積もり, 7

構造体, 38

サブルーチン, 28

実績, 8

車輪の再発明, 39

仕様書, 8

定数化, 36

ドキュメンテーション, 19

名前空間, 38

人月計算, 7

マジックナンバー, 36

予実管理, 8

# 筆者自己紹介

野村 周平　P/N：眞久 秀（略してまく）

三十路間近の公私共にプログラマ。中学校の部活動で触らされたN-88日本語BASICがきっかけで、それ以来今に至るまでどっぷりとゲームプログラミングの世界に入り込み、同人やフリーでいくつかの作品をリリースした。

業務での開発歴は五年程度。ウェブアプリとPCゲームを2:1位の比率で開発してきた。副業で専門学校にてゲームプログラミングの講師をやっている。こちらはまだ始めて一年の新人です。

筆者Webサイト「danmaq」  
http://danmaq.com/

"大きなゲーム"製作 プログラミング入門

2011年3月28日 初版発行

著者：野村 周平

発行元：danmaq

東京都墨田区東向島2-36-12

オリエントビル57 1115号室

内容の不明点などご意見は  
メールでお気軽にどうぞ。

info@danmaq.com

1. 初心者向けプログラミング解説サイトとして有名。 http://www.kumei.ne.jp/c\_lang/ [↑](#footnote-ref-1)
2. 人日や人月の方が一般的です。人時はむしろマイナーでしょう。 [↑](#footnote-ref-2)
3. 実際はメンバーの能力差もありますし、お互い認識の誤差もあるでしょう。それを埋めるために打ち合わせが要りますし、そうすればその分の時間のロスが生じます。人月計算はメンバーが少ないほど、正確に出やすくなります。 [↑](#footnote-ref-3)
4. 教えておきながらなんですが、私個人的には人月計算は大嫌いです。  
   もしメンバーが2100人いたら1分でゲームが作れますか？ [↑](#footnote-ref-4)
5. 下から突き上げると画面中の敵が全員ひっくり返る特殊な床です。 [↑](#footnote-ref-5)
6. 実際仕事でやる場合は、もう少し水増ししたりします。それぞれの機能を繋げるのにも時間がかかりますし、複数人数でやる場合は打ち合わせによるロスもありますし。経験上一人で作る場合は2倍、数人で作る場合は3倍くらいにすると後々言うこともなく快適に開発が進められます。 [↑](#footnote-ref-6)
7. ここでは実装だけの工数だけを説明しています。実際にはグラフィック製作やサウンド製作、テストプレイやマニュアル作成、果ては特設ウェブサイト製作も工数に含みます。 [↑](#footnote-ref-7)
8. あまり大きなゲームでこれをやると、筆者ですら発狂しそうで怖い。 [↑](#footnote-ref-8)
9. 実際、これを書くだけでも結構疲れました。 [↑](#footnote-ref-9)
10. コメントの一種だが、専用ツールを使用してメソッドやフィールドと紐づけたコメントを文書化できる特殊なコメントのこと。JavadocやPODなど言語ごとに実装は異なる。 [↑](#footnote-ref-10)
11. 本当に理想であり、夢物語なんですけどね。1分ならまだ望みはあっても、3秒は流石に……。 [↑](#footnote-ref-11)
12. 以前は1行80桁まで、とよく言われていましたが、流石に今時では短すぎるでしょう。ちなみにMS-DOSやN-88 BASICの画面が横80桁だったところから由来していたりします。 [↑](#footnote-ref-12)
13. 勇気ある方は完成版をいきなり見てみるのも良いでしょう。 [↑](#footnote-ref-13)
14. 「フォントサイズを小さくすれば良い」と答えた方がいました。折角なのでユーモア賞としてここに掲載しておきましょう。 [↑](#footnote-ref-14)
15. 本書の中盤～後半で詳しく解説しますが、そのクラスのインスタンスが、一つないし固定数存在することが保障されるアルゴリズムのことです。 [↑](#footnote-ref-15)
16. 2008年頃にそういう現場の経験もあります。ちなみにOSはWindows2000でした。Word97やExcel97を使っていて、96年頃のLotus Notesを使っていました。堂々とNIHを謳っていた割に、この辺は別にいいのかなぁ。どうでもいい雑学でした。 [↑](#footnote-ref-16)
17. これを「四角い車輪の再発明」という人も、少なからずいるようです。 [↑](#footnote-ref-17)