インタラクティブゲーム制作 ゲームプログラミング講座 第13回資料

竹内 亮太

(2010/7/21)

13　状態を制する者はゲームを制す

今期の最後は、ゲームプログラミング (実はゲームに限らないですが) で理解するべき最重要事項の一つ、状態遷移について詳しく掘り下げます。これ以外にも大事なことはたくさんあるのですが、後期に取り扱うことにします。

13.1　状態のレベル分け

状態遷移については、言葉だけなら2回目あたりで出したと思います。ゲームプログラムはぐるぐるループしながら状態が移り変わっていく構造になっているため、**今が"何状態"なのか？**という状態の把握と**どういう時に状態が移り変わるのか？**が非常に重要になっていくわけです。

ここでは私が今でも大好きな、某格闘ゲームを例に挙げて考えて見ましょう。

13.1.1　シーン状態

格闘ゲームに限らないかもしれませんが、ゲームは複数の画面を持ち、プレイヤーの操作によってその画面を行き来することが多いです。ゲームプログラミングにおいては、1つの画面のことを「シーン」と呼びます。FK には fk_Scene というクラスがありますが、それとはまた別の概念だと考えてください。

某格闘ゲームでは以下のような画面モードが存在します。

1. メーカーロゴ・OP ムービー・タイトルロゴ
2. メニュー
3. キャラクターセレクト
4. バトルシーン (当然一番のメインになる)
5. イベント・エンディング・ゲームオーバー

私が某格闘ゲームを作るなら、このくらいのシーンに分けて考えます。もう少し統合してもいいかもしれませんけどね。

で、賢明な皆さんなら「これらを構成するソースコードが全部 main() に入っていたら」どういうことになるか、もうおわかりですね？当然そんな組み方はしない訳です。まっとうに組むのであれば、一つ一つのシーンをクラス化し、シーンの移り変わりに応じて対応するシーンクラスの処理を呼び出す、というふうにしていくことになります。

fk_Scene は、あくまでその画面で表示するもののリストを管理するのが主な仕事なので、上記のような目的を達成するには、

- そのシーンで表示する形状
- そのシーンで使う fk_Model
- または上記2つを含んだ自作クラス
- 上記3つを制御する処理

などをクラス化する必要があります。

とはいえ、最初からこれらをみっちり考えていては手が止まってしまいます。なので皆さんは「どこか1つのシーンを1つのプロジェクトで作ってみる」ところから始めることをおすすめします。ゲーム全体をいきなり作ろうとすると骨が折れますので、まずはゲームのメイン部分だけでも動かせるようにしてみると良いでしょう。もちろん「俺はメニュー画面を作るのにコーフンしちゃうんだ」などという変わり者もいると思いますので、そういう人はその部分だけ作ってみる、のもありです。

1つのプロジェクトで画面を作る時でも、一応クラス化のことを頭に片隅に入れておくとよいでしょう。うまくいけば、クラスに移植してちょろっと手直しするだけで複数のシーンがガチャーンと合体します。これができると、複数人での分担もしやすいというものです。

13.1.2　キャラクター状態

さて、おおざっぱな画面の構成や流れも状態の一つですが、それより細かいレベルでの状態遷移も考える必要があります。ゲームのメイン部分における、キャラクターの状態遷移です。ここでも某格闘ゲームを例に挙げて考えて見ます。

題材にするのは某格闘ゲームの某主人公です。彼の持っている技は、

- 近距離立ち技 (パンチとキック・弱中強で 6 種)
- 遠距離立ち技 (パンチとキック・弱中強で 6 種)
- しゃがみ技 (パンチとキック・弱中強で 6 種)
- 垂直ジャンプ技 (パンチとキック・弱中強で 6 種)
- 斜めジャンプ技 (パンチとキック・弱中強で 6 種)
- 地上特殊攻撃 (2 種)
- 共通システム特殊攻撃 (3 段階)
- 投げ技 (2 種)
- はどーけん！(弱中強 EX で 4 種)
- しょーりゅーけん！(弱中強 EX で 4 種)
- たつまきせんぷーきゃく！(弱中強 EX で 4 種)
- 空中たつまきせんぷーきゃく！(弱中強 EX で 4 種)
- しんくーーーはどーけん！(弱中強で 4 種)
- めつ！はどーーーけん！

これだけあります。これらは全て「固有の状態」であり、個別に処理が必要です。しかもこれに加えて、

- 立ち状態
- しゃがみ状態
- 垂直ジャンプ状態
- 斜めジャンプ状態
- 移動状態
- ダッシュ状態
- 地上やられ状態
- 空中やられ状態
- ダウン状態
- ヘブン状態 (KO されてると思いねぇ)

など「技を出している」以外の状態も加わり、その遷移の管理はまさにカオスの様相を呈します。

状態遷移で重要なポイントは**どの状態からどの状態へ移行するのか何がきっかけになって状態の移行が発生するのか**をしっかり管理することです。たとえば、ジャンプ中に何もしなければ、そのまま着地して立ち状態に移行しますが、ジャンプ中に強キックボタンが押されたら、当然ジャンプ攻撃固有の状態に移行します。それが済んで着地するとまた立ち状態に戻るわけです。

しかし、格闘ゲームの場合は相手に攻撃を当てられると、途中で強制的に「やられ状態」への移行が発生します。このような割り込みのコントロールを考慮する必要があります。やられ状態への移行が発生しない状態は、すなわち「無敵状態」にあたるわけですので、ダウン中に無制限で攻撃できてしまうのを防いだり、一部の強力な技に「やられない時間」というのを設定することになるわけです。

また余談ですが、格闘ゲームにおける「キャンセル」という動作も、状態遷移で説明することができます。攻撃を当てた後の「立ち技硬直状態」の時間に必殺技のコマンドが完成したら、強制的に必殺技の状態へ移行する、というふうに仕込めばいいわけです。

13.2 状態遷移の実践

では某格闘ゲームの主人公ほどではないにせよ、複数の状態を持ったキャラクターをプログラムしてみましょう。「FK Performer のモーション再生クラスと使用サンプル」を見てみてください。

このサンプルには motionID という変数が存在し、その変数の値に応じて再生するモーションや、対応する処理を切り替えています。私はこの「1 モーション 1 状態」で対応づけて管理するのが基本ではないかと考えています。複数のモーションを一つの状態にまとめてしまうと細かな操作感が出せないので、動かしていて気持ちよくないです。

しかし、この数値で状態を管理していくのはなんとも言えない気分になってくると思います。そこで皆さんには、古くから使われている「数値への命名方法」と、通好みの条件分岐である「switch-case 文」を伝授したいと思います。

13.2.1 列挙型

現在のモーション管理方法では、0 は歩き、1 はパンチ、2 はキック、という対応付けをきちんと覚えていなくてはなりません。しかしそんな管理方法では、モーション数が増えていった時に確実にやっていられなくなります。

そこで皆さんに伝授するのが **enum 型**と呼ばれる「数値への命名方法」です。まずはこちらをご覧下さ

い。

```
enum Girl_MotionStatus {
    GIRL_STOP = -1,
    GIRL_WALK,
    GIRL_PUNCH,
    GIRL_KICK
};
```

これを宣言しておくと、カンマで列挙した名前に対して順番に整数値が定数として割り振られます。GIRL_STOP が-1 として、それ以降の名前は 0,1,2... という数値の代わりとして使えるわけです。つまり、モーションの切り替えを指示している部分がこのように書けます。

```
    // 変数宣言時の初期値がこんな感じ
    int motionID = GIRL_STOP;

(中略)

    // キー操作によるモーションの状態変化
    if(fk_win.getKeyStatus('1')) {
        motionID = GIRL_WALK;
    } else if(fk_win.getKeyStatus('2')) {
        motionID = GIRL_PUNCH;
    } else if(fk_win.getKeyStatus('3')) {
        motionID = GIRL_KICK;
    } else if(fk_win.getKeyStatus('0')) {
        motionID = GIRL_STOP;
    }
```

どうでしょうか？数値がうろ覚えになりながら作業することを思えば、断然やりやすくはなっていると思います。注意するべきは、列挙順とモーションの読み込み順が揃ってないと詰む点ですかね。私も何回もやりました。

enum の書き方について少し補足します。「列挙名 = 0」のようにすることで、割り振る数値を仕切り直すことができます。仕切り直した後は+1 ずつされていきます。最初に何も代入しなかった場合は 0 から順番に割り振られていきます。

enum の後ろに「Girl_MotionStatus」という変数型名っぽい記述がありますが、これはそのまま変数の型として使えます。つまり、その enum で列挙した定数値のみをやりとりすることができる変数というのを作ることができるわけです。

```
    Girl_MotionStatus motionID = GIRL_STOP;
```

このように扱えます。通常の int 型の変数で扱っていると、定数名以外の代入なども受け付けてしまうので、せっかく enum 型を作ったのならその型の変数を利用した方が安全かもしれません。int を引数に取る関数に列挙型の値を渡した場合は、そのまま自然に int 型の数値であるかのように扱われます。

13.2.2 switch-case

さて、では条件分岐も if 文の条件式に列挙型使って解決、で済ませたいところですが、今回のように「ある値の状態に応じて処理を分岐させる」という場合においては、正直に if-else if-else if... と列挙していくのはうまい方法ではありません。

多少細かい話になりますが、条件分岐はコンピュータにとって結構重たい処理になります。今回のように、単一の整数型変数が持っている値に応じて処理を分岐させたい場合は、switch-case 文が便利です。

```
    switch(条件分岐で参照する変数) {
        case 値 1:   // <-コロン (:) です
// 値 1 の場合ここの処理を実行する
            break;
// break を付けないと、下の処理まで流れていっちゃう
        case 値 2:
// break を付けずに次の case を書けば、
// 両方の case で共通の処理が書ける
        case 値 3:
// なのでここの処理は値が 1、2 だった場合に実行される
            break;
        default:
// 上までで書いた case に当てはまらなかった場合の処理
// なくてもいいが、書かないと警告が出る場合もある
            break;
    }
```

使い方はこんな感じです。整数型限定、case のところに記述する値は定数値でないとだめ、などと制約も多く、break を付けないと処理が流れていってしまうことから、世の中には switch 文を毛嫌いする人もいます。ですが私は結構好きです。列挙型と組み合わせた時の見通しの良さはなかなかのものだと思います。意図しない break 忘れは怖いですけどね。

switch 文を使うもう一つのメリットとして、処理の高速化というのがあります。例えば 10 個の状態値についての分岐を書いた場合、if-else if で書いていくと条件判定処理 (条件式の中身について true か false かをチェックする処理) は最悪の場合 9 回発生します。ところが switch-case は「その変数の値な〜んだ？じゃあその case へ飛べ！」で済むので、「この条件？違う…この条件？違う…この条件？」とやっていくのに比べると遙かに高速です。まぁ最近の CPU では誤差レベルかもしれませんけどね。

とはいえ、プログラムがどういう風に動作しているかをイメージすることは、プログラムセンスを磨く上で重要ですし、たとえ誤差程度の速度差でもその処理を何千回、何万回と通過した場合の時間の差は無視できないものになったりします。if と switch の違いは、コンピュータアーキテクチャとプログラムの関係を見直すきっかけになるネタだと思います。

13.3 もっと効率的な状態遷移

今回はとりあえず原始的な方法で状態遷移を考えてみましたが、実際のゲームではもっと複雑に状態が絡み合ってきます。これをうまくさばくには、**階層的な状態遷移構造**を考える必要があります。某ゲームで言えば、「地上・空中」で大枠を分類し、さらにそれぞれ「立ち・しゃがみ」と「垂直・斜め」で分けていく、というような感じです。

ゲームをデザインする時は、このような状態をリストアップし、どのように繋げて管理していくかを必ず考えてまとめてから作業に入るようにしましょう。これは仕様書の一部としても重要な要素です。たとえ後から追加や変更があったとしても、それを行き当たりばったりでコーディングするのと、リストなりフローチャートなりで管理しながら進めていくのとでは、効率に雲泥の差が出ます。

最初のうちは少ない状態をシンプルに遷移させるだけで構いませんが、大がかりになってきそうなときには「階層的に分類できないか」「キャラクター固有の状態と、共通の状態に分類できないか」という点を考慮しながら進めてみてください。

13.4 終わりに

さて、今期の間に色々なトピックを学んできましたが、いかがでしたでしょうか？「今年は 2 年目だし、資料の使い回しでいいから楽できるぜ〜」と思っていたのも今は昔、去年の後期やったことと被らないように気を遣ったり、コードレビューという新しい試みもあったりで、なかなか大変でした。皆さんの環境との微妙な違いで不便をかけたりもしましたし、まだまだ発展途上であることを痛感しました。

そんな授業でしたが、皆さんは何を学んだでしょうか？この授業では、必ずしも「これをできればプロになれる」「この方法が一番効率が良い」という最短ルートを提示するものではありませんが、こちらの想定する皆さんの段階から、次へとつながるエッセンスをより分けてお届けしてきたつもりです。

この授業の内容をどのように繋いで形作っていくかは、皆さん次第です。夏休みを有意義に使って、お互い大幅にパワーアップした状態で、後期を迎えたいですね。3 年生の皆さんにとっては正念場が続きます。若いうちは「無茶」するのもいいですが、「無理」はほどほどにしてくださいね。いのちだいじにしつつ、ガンガンいっちゃってください。

13.5 課題提出について

では課題提出のお話をもう一度しておきます。**提出期限は 8/11 23:59** です。Assit の MPPF での提出になります。提出欄を見てもらえばわかるように、出せるファイルは 1 つだけなので、提出したいフォルダを 1 つの zip ファイルにまとめて圧縮してください。

提出対象となるのはこれまでの課題すべて (第 2 回を先行提出している人はボーナスが付きます。今回の提出で改良が加わっている場合は更に上乗せします) と、「その他」と称しまして、自由に作った何かを提出してもらえる枠を用意しました。これは主に 2 年生を対象としていて、どの課題、というわけではないけれども作ってみたものがある場合、それをもって自分の学びっぷりを見てほしい場合は、どしどし提出してください。ただし、実行もコンパイルもできないのは困りますので、そこは適宜ドキュメントを添付するようにしてください。

提出はできる範囲のもので結構です。無理して全部に手を出すより、ある部分について掘り下げた課題というのも十分評価に値します。もちろんガッツのある人はそれを見せつけてくれても構いません。