No.2 応用 一般

いちからゲームを作ってみよう！ ―ゲームプログラムの基礎―

安原祐二（ユニティ・テクノロジーズ・ジャパン合同会社）

ここではゲームを「インタラクティブなアプリケーション」とする．すなわち，入力に従って応答があるソフトウェアだ．ゲームには長い歴史があるが，それを実現するプログラムの中心部分は意外なほど変わっていない．今回はそんなゲームプログラムの基本的な実装を，順を追って説明していく．言語はC++に準じているが，ビルドして実行するために必要な細かな記述は省略されている．よってそのままで動作するものではないが，そのエッセンスを受け止めていただければ幸いである．

メインループ

初期化と無限ループ

```
int main()
{
    initialize();
    for (;;) {
        update();
    }
}
```

program 1

まずはmain関数から．実行直後は初期化initializeである．そのあとはループfor (;;)に入り，もう抜けることはない．ループごとにupdateが呼ばれ，ここにゲームの更新処理を書く．このループ1回をフレームと呼ぶ．このような初期化とループの記述は多くのアプリケーションで共通の仕様であり，この段階ではまだゲームの特徴は現れていない．

同期待ち

```
int main()
{
    initialize();
    for (;;) {
        update();
        waitVSync(); // 同期待ち
    }
}
```

program 2

同期待ちをする関数の名前は，waitVSyncとする．モニタの垂直帰線信号を待つわけだが，簡単にいえば，一定時間が経過するまでCPUをスリープさせるのだ．

同期待ちはとても大事な概念で，これによりupdateの処理時間が早かったり遅かったりしても，ループの頻度が安定する（図-1）．

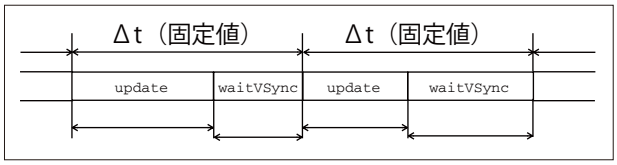

図-1　フレームを安定させるwaitVSync

固定周期でループさせるのは，滑らかな映像を作るためにはとても大切なことだ．勘違いしている人がいるかもしれないが，フレームレートが高ければ高いほど映像が滑らか，というわけではない．もちろんそれも大事な要素だが，周期が一定であることも重要なのである．もちろんゲームの種類にもよるが，たとえば1フレームの更新に要する時間が16msと33msを交互に繰り返すと，結果的に秒間40フレームになる．が，それよりも33ms固定にした秒間30フレームの映像のほうが滑らかだ．これは筆者の主観ではあるけれど，ぜひ実験で確認して

いただきたい．

さて，そんな大事な waitVSync だが，その名のようにモニタの垂直帰線を待つ実装とは限らない．単純にタイマーで実装することも可能だ．大事なのは，update の処理時間に変動があっても，そのタイミングを固定に保つ機構ということだ．

◈ 入力

これがないとインタラクティブにならない．キーボードやマウス，あるいはパネルへのタッチ情報，ゲームパッドの入力．これらの情報を取得する部分になる．

```
int main()
{
    initialize();
    for (;;) {
        input(); // 入力処理
        update();
        waitVSync();
    }
}
```

program 3

input で入力情報を受け取る処理を記述する．ほとんどのゲームプログラムは，「イベントのハンドラを実装」という形では実装されない．こうやって毎フレーム入力情報を受け取っておき，input の内部でどのキーが押されているかの情報を保持しておくのだ．こうすることで，押しっぱなしや複数キーの同時押しなどの情報をゲームに応じて準備することができるようになる．これは一般のアプリケーションとゲームとの一番の違いと言ってもよいだろう．できるだけ生に近いデバイスの情報を扱うことで，ゲームに応じた最も快適なインタフェースを実現するのだ．

さて，この例では update の直前に input を書いている．ここで，update の中に input を記述するのはまずいやり方と覚えておいてほしい．update はあくまで更新処理であり，入力とは分けることが大事なのだ．たとえば早送りなどで update を複数回ループさせたいことがあるかもしれない．そんなときデバイスに依存する input もループさせてしまうと，デバイスからの情報取得のタイミングとは異なる周期になるので，高い確率で問題が発生することだろう．

◈ 描画

そして描画，すなわち画面の更新だ．

```
int main()
{
    initialize();
    for (;;) {
        input();
        update();
        render(); // 描画処理
        waitVSync();
    }
}
```

program 4

この render 部分で，画面にグラフィクスを表示させる．ここでも update の中に render を書くべきではなく，update にはゲームの情報（たとえばプレイヤの位置など）の更新に専念させ，表示は render に分けるようにする．こうすることで，効率の面でもプログラムの見通しという面でも，良い結果に繋がる．たとえばゲームを更新する際，Δt という値を update に渡すことがよくある．これは update 1 回でゲーム中の時間がどのぐらい進むかを表した値だが，この Δt の概念は render とは無関係だ．つまり render はその瞬間の状態を描画することに専念すべきなので，update とは分離するのである．

さて，これでゲームの大枠が完成した．もちろんそれぞれの関数の中身を記述しないとゲームにはならないが，大体どんなゲームでも，メインループはこのように記述する．

次に，update の内容を実装する上で大切なタスク（task）の概念を説明する．

タスクシステム

◈ タスクループ

CPU が一度に実行できる命令は 1 つなのに，ゲームを見ていると画面上をたくさんの物体が同時に動

き回っている．これはどうなっているのだろう？

その答えとして，タスクシステムを書いてみよう．タスクというのは小さな実行単位で，たとえばシューティングゲームであれば自機，敵，弾，そういったものがタスクになる．（なおタスクというのは方言でもあり，開発会社によって異なる呼び方があるようだが，筆者の経験する限り，タスクという呼び方で通じる）

```
void update()
{
   for (auto it = task_manager.begin();
        it != task_manager.end();
        ++it) {
       it->update();
  }
}
```

program 5

メインループのところで呼び出していた update の中身で，タスクシステムの実行（呼び出し）を記述してみた．task_manager というオブジェクトに，各タスクオブジェクトをリストとして登録しておくと，このようにイテレータでループさせることで，それぞれのタスクが持つ update を実行する．つまりこの task_manager はコンテナになっており，複数のタスクオブジェクトを内部に保持しているわけだ．シングルトンとして実装し，アプリケーションが起動している間は必ず 1 つ，存在するようにしておく．

◈ GameObject の実装

task_manager が保持しているオブジェクト群を，GameObject というクラスで実装する．GameObject が多種多様なゲーム要素を実現するための実装にはいくつかの方法があるが，今回はコンポーネントという概念を採用して進めてみよう．つまり複数のコンポーネントを保持したオブジェクトを，GameObject と呼ぶことになる．

```
class GameObject {
   list<Component> component_list_;
   void update();
};
```

program 6

GameObject のクラス宣言だ．このように GameObject は Component のリストを持つ．では update の実装も足してみよう．

```
class GameObject {
   list<Component> component_list_;
   void update() {
      for (auto it = component_list_.begin();
         it != component_list_.end();
         ++it) {
         it->update();
      }
   }
};
```

program 7

便宜上，関数定義を宣言内に記述した．このように，保持しているすべてのコンポーネントの update を呼び出す記述になる．

◈ Component の宣言

その Component は，抽象クラスとして宣言される．

```
class Component {
public:
  virtual void update() = 0;
};
```

program 8

このように「何かしら update するもの」となるわけだ．

これで，タスクの基礎となる構造は完成である．あとは，必要に応じて GameObject を生成し，その GameObject の性質に応じて Component を追加する．もちろん Component は抽象クラスだったので，このまま生成することはできない．作成するゲームに応じて，具体的な機能を別途定義することになる．

入力について

現代では，さまざまな入力デバイスがある．

- キーボード
- マウス
- タッチパネル
- 加速度センサ
- ゲームパッド
- カメラ（ジェスチャを識別するなど）

まだまだあるだろう．脳波など．汎用的なゲームエンジンを作る場合は，ありとあらゆる入力を扱える必要があるが，ここではとりあえずマウスだけを考えてみよう．

イベントハンドラではない

ゲームをプログラミングする場合の大事な考え方は，入力をイベントとして扱わないことだ．先に説明したようにフレームという概念があるので，そのフレームで入力デバイスの状態を取得し，アプリケーション内で使えるように値を保存しておく．

```
int input_x = 0;
int input_y = 0;
long input_bstate = 0;

void input()
{
   MEVENT event;
   getmouse(&event);
   input_x = event.x;
   input_y = event.y;
   input_bstate = event.bstate;
}
```

program 9

input は最初にメインループで作成した関数だった．今回は例として ncurses というライブラリの記述に倣っているが，その瞬間のマウスの位置を取得して保存している，という点に注目してほしい．作成するゲームに応じて，入力デバイスにアクセスする関数をここに書く．

今回はグローバル変数に値を保存したが，きちんと書くならシングルトンオブジェクトを用意し，そこに保存するのがよい．ともかくこれで，input_bstate を参照すればいつでも「そのフレームの」マウスの状態を取得できるようになった．マウスカーソルの位置に加えて，ボタンを押し下げているか否かの情報も保存してある．

このようにフレームごとに値を保存すべきであり，使用する実装の中でデバイスの状況を取りに行ってはならない．値が更新されるタイミングを確定させないと，思わぬ不具合を引き起こすだろう．

また，なるべく入力はハードウェアの生の状態を扱えるようにしたほうが，ゲームの表現が広がる．たとえば，押しっぱなしにすることに意味があるゲームデザインにする場合，押し下げと押し上げをそれぞれのイベントで表現するのではプログラムが不必要に煩雑になってしまう可能性がある．もちろんアプリケーションの仕様が固まってからは，その仕様に応じて入力処理で加工しておくのは良い考えとなるだろう．

レイテンシ（遅延）

余談となるが，ゲームの中には本当に入力にシビアなものがあり，その場合は実際の入力（指がボタンを押し込んだ瞬間）とこの input 関数でその値を取得するまでの期間を短くするための工夫が必要になる．この期間を入力レイテンシと呼ぶ．筆者はレーシングゲームを製作した経験があるが，入力レイテンシが大きくなった（つまり遅れが生じた）際，ドライビングテクニックが優れたプレイヤにはすぐに気付かれて「これでは良いタイムを出せない」と指摘される．

実際には render で描画した映像がモニタに表示されるまでの時間もレイテンシとして計上され，近年ではこちらの比重の方が大きくなっている．最近のテレビには画像モードに「ゲームモード」を備えているものが多いが，大抵の場合それは映像レイテンシを小さくするモードとなる．こういったテレビには遅延を大きく取ることで映像を美しく加工する技術を搭載しており，ゲームのようなインタラクティブコンテンツのためにそれを無効にするオプションを用意している，というわけだ．

ゲームエンジンへ

さて，ここまでですでにゲームプログラムの基本は整っている．図 -2 で示されているものだ．

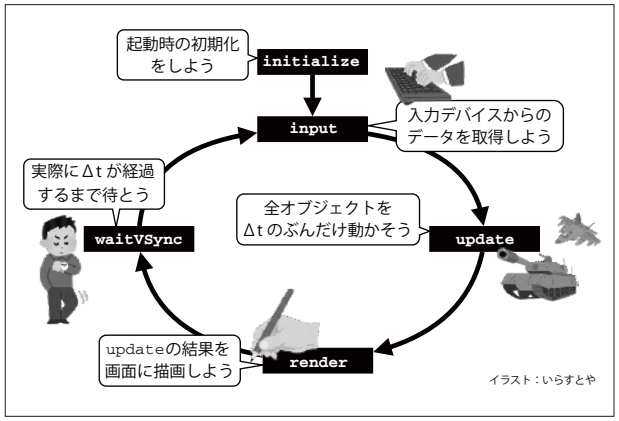

図 -2　初期化とメインループ

あとは作るゲームに応じて

・ゲームロジックを記述する

・ゲーム内容を描画する

というのを足していく．たいていのゲームではゲームロジックにはコリジョン，すなわち衝突判定をする部分が必要だろう．衝突判定後の挙動も含めて，これは物理エンジンと呼ばれるものだ．また，描画には OpenGL や DirectX などの描画ライブラリを呼び出す部分が必要になる．これはレンダリングエンジンだ．

一般的にゲームエンジンと呼ばれるものは，こうした物理エンジンやレンダリングエンジン，そのほかゲームに必要なさまざまな要素を備えている．それらはゲームによって必要であったりそうでなかったりするものだが，今回紹介したゲームループはその中心となるもので，おおよそすべてのゲームで共通の要素となるだろう．

ところでゲームプログラムには，もう 1 つ重要な要素がある．データだ．次はそこに触れる．

データ

ちょっとしたゲームを作る際でも，データは必要になってくるだろう．すなわち，

・3DCG のための頂点，およびテクスチャ

・タイトル画面やボタンを表現するための画像

・コリジョンの情報のためのジオメトリ情報

・レベルデザインにおける配置情報

などだ．これらの情報をソースコードに埋めるわけにはいかない．効率が悪いし，何よりもチームでの共同作業ができない．これらのデータを生成するのも，またそれを読み込むのも，ゲームプログラムの大切な要素となる．

◈ バイナリかテキストか

データフォーマットをバイナリとするか，またはテキストにするかはそれぞれ一長一短がある．ただ，商用のゲームを作ろうと思った場合は，通常はバイナリを選択しておくものだろう．大雑把にいえばこれは，作業時の効率を犠牲にして実行時の効率を高める，という思想になる．もっとも，近年は実行効率よりも作業効率の価値が高いケースも少なくはない．実行効率を重んじる考え方は，どちらかといえば古い思想かもしれない．

◈ シリアライズか直接マッピングか

また，データをロードする方式としてシリアライズとするか，直接マッピングするかの選択がある．メモリ効率を最大化しようと思うと直接マッピングを選ぶことになるだろう．これは前項でバイナリを選択した場合にのみ可能になる．今回は実行時の効率を重視し，データを直接マッピングする手法を説明していく．

◈ データ生成ツールの役割

```
struct Data {
   long data0;
   Data* next;
};
```

program 10

program 10 における構造体 Data は，int 型のデータ data0 に加えて自身のポインタ next を持っている．これで連結リストとなるので，可変長の連結データを表現できる（ある程度規模の大きいゲームを作る場合，データが可変長であることは効率の面から欠かせない）．ゲーム側，すなわちロード側は，

特定の連結データをファイルから読み込み，プログラムでは同じ Data 構造体でアクセス可能となるのを目的とする．

このデータを生成するために，データ生成用のツールをゲームとは別に作成することになる．今回はそのプログラムは省略するが，生成はどんなプログラムで記述してもよい．ゲーム開発において，データの生成はターゲットマシン以外の環境用のプログラムを書く代表的な例といえるだろう．状況に応じて最適なプラットフォームを選択することになるのだが，幅広い知識や経験があればあるほど，最適な環境を選択できることになる．

さて，具体的なデータで説明しよう．たとえば図 -3 のようなデータをゲームで使用したいとする．

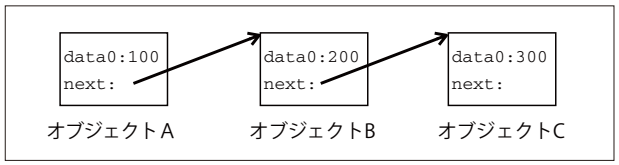

図 -3　連結されたデータの例

3 つの Data オブジェクト A，B，C は next 要素によって連結されており，それぞれ 100, 200, 300 という値を保持している．この連結された状態をデータで用意しなければならない．

data0 に入れる値はそのまま 100 などの数値を入れればよいので分かりやすい．だが next に入れる値はどうしたらよいだろう．単に 3 つ並べるだけでは，より複雑な連結を表現する際に破綻してしまう．

ここは，すべてのデータが連続してメモリに配置されることを期待して図 -4 のようなデータを生成する．

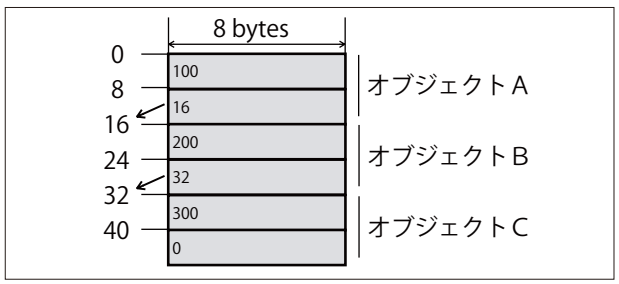

図 -4　生成されるデータ

8byte の項目が 6 個で，合計 48bytes のデータとなる．このデータを作るためにまず必要なのは，ゲームを実行する環境での struct Data のサイズになる．今回は 64bit 環境を想定するため，long も Data* も 8bytes となる．よって sturct Data のサイズは 16bytes だ．必要な Data オブジェクトは 3 つなので 3 つ並べ，data0 にはそれぞれ希望の値（100，200，300）を入れておき，next にはこのデータの先頭からのオフセットを入れる．つまり，オブジェクト A の next にはオブジェクト B の先頭からのオフセットが 16bytes なので 16 を入れ，オブジェクト B の next にはオブジェクト C の先頭からのオフセットが 32bytes なので 32 を入れる．

このデータをファイルとして吐き出せるようになったら，データ生成ツールとしての最低限の役割を果たせたことになる．実際の開発環境では，データ生成ツールは必要に応じて，こうしたデータをチームメンバが作りやすいように GUI で実装する必要があるだろう．

◈ データ読み込み側の処理

次に読み込み側．最初に書いたプログラムの for(;;) に入る前に，initialize(); という関数があった．ここにデータを読み込む処理を書くことにしよう．

```
void initialize()
{
   void* buf = readFile("data.bin");
   Data* data = reinterpret_cast<Data*>(buf);
   data->map(buf);
}
```

program 11

readFile は与えられたパスのファイルサイズでメモリを確保し，そこにファイルの内容を読み込む関数とする．readFile に特定ファイルのパスを渡すことで，そのファイルの中身が buf が指し示すメモリに格納された．先ほど作成したデータは 24bytes だったので，確保されたメモリも 24bytes となる．

そして buf は（強引に）Data* にキャストされる．さらに data はオブジェクトとして振る舞うために，map メソッドをコールする．

◈ map メソッドでポインタの整合性を保つ

Data は内部でポインタを持っているので，ただ

読み込んでキャストしただけでは正しく動作しない．そこで map メソッドでポインタとして振る舞えるように，実際のメモリアドレスに変換する．

```
void Data::map(void* buf)
{
   if (!next) return;
   next = reinterpret_cast<Data*>(
     reinterpret_cast<long>(next) +
     reinterpret_cast<long>(buf));
   next->map();
}
```

program 13

コンパイルエラーを出さないために reinterpret_cast を何度も呼んでいるが，アルゴリズム上の意味はない．この行だけを分かりやすく記述すると次のようになる．

```
void Data::map(void* buf)
{
   if (!next) return;
   next = next + buf; // 実質的な動作
  next->map();
}
```

program 14

ファイルから読み込んだ直後の next には先頭からのオフセット値が格納されていた（前述のデータ生成についての記述を参照されたし）．読み込み後のメモリの先頭アドレスは buf なので，buf に オフセットの next を加えたものが，求める next（連結データの次のオブジェクト）の実アドレスとなる．実アドレスとなった next を用いて，次のオブジェクトの map を呼ぶ．ここでは再帰呼び出しとなっており，最後には next は 0 が格納されているため終了となる．

これで読み込んだデータはプログラムで使用できる，実アドレスを保持するようになった．この map メソッドは，実アドレスへの解決の役割を持つ．なお，このように単なる数値を実際のアドレスに変換する操作は，実行する環境に強く依存する．reinterpret_cast という記述はメモリの整合性を損なう可能性があるので，注意深く使用する必要がある．また，複数回の map コールでも容易に破壊されるため，データには 1bit 以上の領域をあらかじめ用意しておき，map 呼び出し時にアドレス解決済みを示す値を書き込んでおき，2 回目以降の map を抑制するのがよい．

◈ シリアライズとの比較

現代ではデータを生成する際に，ここで述べたような直接マッピングではなくシリアライズを採用するのが一般的かもしれない（読み込む方はシリアライズの反対なのでデシリアライズ）．良い点として，環境に依存しにくい．reinterpret_cast のような強引な手法を使わなくてもデータを読み込めるため，同じデータをさまざまな環境で使い回すことも期待できるのだ．ただしデメリットとして，処理効率やメモリ効率が悪くなる．せめてファイルからストリーミングでデシリアライズできればよいが，構造が複雑になるとどうしても，いったんすべてをメモリに読み込んでからデシリアライズすることになる．そうなるといったんメモリに読み込む分余計にメモリが必要となるし，メモリに余裕があったとしても断片化の要因になる．そしてもちろん，読み込み速度の面でも不利になる．

最後に

今回はゲームプログラムの基礎ということで，ゲームプログラムの特徴的な部分，なおかつ最小限の記述について解説した．データの項目を除けば，おそらく拍子抜けするほど単純な記述になっているものと思うが，ゲームエンジンが半ば常識化しつつある昨今では，軽視されがちな部分でもあるだろう．ゲームエンジンに接する際でも，ここで解説した概念を意識しながら扱うことで，格段に使いこなせるようになるはずである．ゲームプログラムを理解する一助となれば幸いだ．

（2016 年 7 月 10 日受付）

安原祐二　■ yuji@unity3d.com
現在はゲームエンジン・ユニティの普及や開発に精を出す日々．ゲーム制作歴は 20 年以上で，さまざまなゲームタイトルの制作に従事してきた．趣味は将棋，特技はジャグリング．娘よりも妻が好き．