ライブラリ概要

平山 尚

2008 年 9 月 2 日

1 構成

現状の構成は以下のようになる。

Framework　モジュール統括、ゲームループ、ウィンドウ
WindowCreator　ウィンドウ生成、メッセージループ
Scene　上位グラフィックス
Graphics　下位グラフィックス
Sound　音
Input　キーボード、マウス、ジョイスティック
PseudoXml　Xml ライブラリ
FileIO　ファイル IO
Threading　スレッド、同期
Math　乱数、ベクタ、行列、超越関数
Base　データ構造、メモリ管理、その他ユーティリティー

依存関係を図にすることはしない。内部で隠蔽されるモジュールは存在せず、全モジュールはユーザから使用可能であるためである。階層で分割されていると理解するよりも、機能で分割されていると理解した方がわかりやすかろう。

なお、現状のモジュール分けは暫定的なものである。例えば Scene 内にある衝突処理ライブラリは、拡充してくれば Geometry モジュールとして独立しうる。アニメーション処理も同様である。Graphics モジュールはシェーダ等の GPU 機能をより多く公開するようになればグラフィックス API の薄いラッパ部分と、その上位のシェーダ管理等を含む上位層に分離するかもしれないし、しないかもしれない。また、Scene モジュールはシェーダ等が入ってくると現状の仮実装では不適切になるため、フォントやプリミティブ描画などの便利クラスを残して削除され、ゲーム側での実装、あるいは一つの標準実装としての外部ライブラリ化がなされる可能性もある。マテリアル、バッチなどの標準的な概念は「並のゲーム」を作るには向くが、GPU 機能を積極的に使おうとした場合にはそぐわないケースが想像されるため、であれば最初からライブラリのコアとしては用意せず、標準的なものをサンプル的に外部に用意した方が理にかなうだろう。

また、動画再生が入ってきた場合は Movie モジュールが追加されるであろうし、PseudoXml は汎用バイナリフォーマットなどを設けた後では Data 等に名前を変える方が良かろう。現状は書籍の話の流れに従った形でモジュール分けがなされており、実用化のために相当の拡充を行えば自然と違った形に落ち着くことになる。

しかし、モジュールの相関関係の大まかなところは変わらないし、各モジュールにしても、また全体としての GameLib ライブラリにしても、後述する設計思想によって構築されていることに変わりはない。

2 設計の特徴

このライブラリの設計は、以下のようにまとめられる。

1. 必要なものだけを見せる
2. 便利さよりも学びやすさ
3. 参照カウントの全面的な採用
4. ライブラリ全体としてどう見えるかを重視

それぞれ説明しよう。

2.1 必要なものだけを見せる

これは、カプセル化の徹底ということである。このライブラリはユーザが直接触りうるクラス以外は一切公開しないし、ユーザが知らなくて良い関数や変数もギリギリまで非公開にしてある。ほとんどのクラスでは private 部は実装クラス内部に隠蔽し、例えば Mutex クラスの private 部は以下のようになっている。

```
class Mutex{
private:
   class Impl;
   Impl* mImpl;
};
```

実際にどう中身が作られているのかは全て Impl クラス内に隠されており、実装が変更されてもユーザに影響は及ばないし、ヘッダのインクルードも少なく、コンパイル時間も軽減される。ユーザに対してライブラリが依存する別のライブラリのヘッダを配布しなくて良いのが最大の利点である。この設計は徹底されており、ライブラリ内のクラスであっても、ユーザに見える必要のないクラスはヘッダが公開されていない。場合によっては cpp 内にてクラスが定義されていたりもする。

2.2 便利さよりも学びやすさ

「いろいろできて便利なライブラリ」よりは、「機能を実現するパーツとしてのライブラリ」を重視し、「あれば便利な機能」の類は最小に抑えた。なぜなら、その手の便利機能はユーザによって使ったり使わなかったりするからである。

あるユーザにとって便利な機能も、別のあるユーザにとっては邪魔な機能にすぎない。クラスや関数の数を可能な限り小さく保つことで、習得コストを低く保つことをより重視した。使用者の大半が使うと思われるものは用意したが、基本的に機能の組み合わせで実現できるものはユーザ側で用意することになる。

使用頻度が低い機能はバグが残りやすく、そのようなコードが大量に存在することはメンテナンスコストも押し上げる。アプリケーションで用意すればそれで済むような処理についてはライブラリ側で持つ必要はない、というのがこのライブラリの思想である。

また、OS あるいはハードウェア的に可能なことであっても、本当に必要かどうかがわからないものについては用意していない。スレッド生成のように、技術のない人が間違って用いた場合に危険な類の機能も極力用意しないようにした。

3 参照カウントの全面的な採用

ほとんどのクラスは参照カウント管理されており、delete による明示的な削除は必要としない。それゆえにユーザのコードが短くなり、バグの発生率も減る。この目的のために、生成は例外なく static な create() 関数によってなされ、通常のコンストラクタは空の参照を作るのみである。

```
{
   Mutex m; //この段階では空
   m = Mutex::create();
   m.lock(); //->ではない。
   m.unlock();
} //デストラクタ
```

Mutex の生成は create() によって成され、ブロックの終わりで自動でデストラクトされる。他の Mutex 型変数に代入された場合は参照カウントが増え、一つが消えてもその場ではデストラクタは走らない。また、ポインタでなく実体でアクセスされるため、「.」演算子でのアクセスとなる。

通常の慣習からはかけ離れた使い方を強いられる点が欠点ではあるが、ポインタ操作によるミス、例えば初期化忘れ、開放忘れ、開放済みポインタアクセス等が原理的に起こらないという利点を重視してこのようにした。

なお、このようなクラスを作るには、クラスの作成時にいくらかの手間をかけねばならず、その点では不利である。実際ライブラリのコード量は増加する。ただし、支援マクロ等によってその手間はかなり軽減されてはいる。

3.1 ライブラリ全体としてどう見えるかを重視

ライブラリに複数のモジュールが存在したとしても、ユーザにとってはライブラリはライブラリである。それらのモジュールが独自の記法や使い方を要求するようなことは極力避け、命名規則や設計の基本はそろえた。

また、モジュールごとに取捨選択して使う、というようなイメージでなく、「一つのライブラリとして扱う」というイメージを優先し、一つの lib ファイルに統合した状態で配布する。何をリンクするかを思い悩む必要はない。

また、各モジュールを統括する Framework モジュールを設け、ウィンドウ生成やゲームループ、終了処理、さらには各モジュールの初期化終了までを一括で行う。このため、ユーザは Framework クラスのいくつかの関数を実装するだけで全モジュールの準備が整った状態でコードを書くことができる。

ユーザが記述するのは以下の 2 関数である。

3.1.1 configure()

起動前設定。解像度その他の設定を行う。

3.1.2 update()

毎フレーム呼ばれるメインループ関数である。ここに来た時には全モジュールが稼動しているため、ユーザが起動処理について考える必要はない。むろん各モジュールの終了処理もこの外で自動的に行われる。

ちなみに、拡張性の軽視も大きな特徴である。拡張可能なつくりは設計のシンプルさや性能に大きな負担をかけ、しばしば使い勝手を損なうものである。するかどうかもわからない拡張のためにそういった短所を引き受けるのは正しいとは言えまい。そのため、ほとんどのクラスは継承による拡張を考えた作りにはしていない。

4 その他の話題

4.1 DirectX

DirectX はバージョン 9c を使っている。うち、D3DX は一切使用していない。これは、可能な限り多くの環境で動くようにするためである。DirectX9 の更新の大半は D3DX に対して行われており、それを使っていなければかなり古い DirectX であっても動作する可能性が高いからである。

なお、シェーダコンパイル関数は D3DX の関数であるため、シェーダは前もってコンパイルしておいて、そのバイナリを文字列定数の形でソースに含めてある。これについては、Graphics モジュールのビルドイベントを参照してほしい。Tools にある ConvertToCode は任意のファイルを文字列定数化するためのものである。

4.2 不足しているもの

実用ライブラリとして用いるためには依然不足している要素が多々ある。また、教育用としても用意しておいて損はないと思われる処理がいくつかある。それらについて詳細に述べることはしないが、ここでは箇条書きにしておこう。

- 容量制限のないタイプの list,set,map
- ソート、検索あたりの標準的なテンプレート実装
- 汎用バイナリフォーマット
- アニメーションのクォータニオン対応
- 自作シェーダを受け入れるインターフェイス
- 画像、音声、動画の独自フォーマット
- 衝突処理の拡充と、幾何計算モジュールの独立化
- 先読み機能付きシーケンス遷移フレームワーク
- sin,cos 等の近似計算のサンプルコード
- 圧縮展開のストリーミング対応
- サウンドの 3D 対応、エフェクト対応、パン用インターフェイス
- 最小限のネットワーク対応

さらに、今後行う予定の改造についてもいくつか触れておく。

4.2.1 Font の Graphics への移動

Scene は windows や DirectX の関数を直接呼ばない場所として作ったが、いろいろあって Font 関連が Scene モジュールに入ってしまっている。これを Graphics に移動したい。

4.2.2 C++/CLI との併用の検討

C++/CLI を用いて中間層を作り、C#でアプリケーションを作るようなケースにも対応したい。そうすれば、ビューアのようなものを作るのが楽になる。ウィンドウ生成とメッセージループを外部から供給できる設計にすることが必要だ。