

マルチプラットフォームライブラリを作ってみた。
PowerSmash4 の場合 (CEDEC2011)

株式会社セガ R&D2 平山 尚

目次

第 1 章	前置き	3
1.1	PowerSmash4 について最低限の紹介	3
1.2	複数機種対応ライブラリという道具	4
1.3	今回開発したライブラリの概要	4
第 2 章	開発時の状況	6
2.1	何故今更作っているのか	6
2.2	使えた資源	7
第 3 章	設計	10
3.1	SegaLib の目的	10
3.2	ライブラリの範囲	11
3.3	関数の集まりか、それともそれ以上のものか	13
3.4	品質のバランス	15
3.5	更新と作業工程	19
第 4 章	実装	23
4.1	ビルド構成	23
4.2	GPU の抽象化	29
4.3	CPU の抽象化	34
4.4	アニメーション	36
4.5	数学	45
4.6	メモリ管理	50
4.7	スレッドと同期	53
4.8	ファイル IO	56
4.9	標準ライブラリや言語機能について	59
4.10	安全性について	65
4.11	XML パーサ	67

4.12	バイナリデータ互換	70
4.13	入力デバイスの問題	73
第 5 章	まとめ	75
5.1	SegaLib のこれから	76
5.2	スペシャルサンクス	77

第1章

前置き

この文書では、PowerSmash4 で使われた複数機種対応ライブラリについて、設計、実装、運用の各面から述べる。

これにあたって、PowerSmash4 というゲームがどのようなものかがわかっていると理解しやすい点が多々あると思われるので、簡単な紹介だけしておく。

1.1 PowerSmash4 について最低限の紹介

PowerSmash4 は、リアル系のテニスゲームである。



以上の画面写真から、このゲームの開発でどんなものが必要になるかはおよそわかっているだけではないかと思う。

今回の話に関わってくる要素を箇条書きにすれば、

- 複数機種対応 (PC,PS3,360,Wii...)
- リアル系であり、画質が求められる。
- 元々アーケードで、開発も元アーケード部署。規模はそれほどでもない (とはいえ 30 人以上)。

今回開発したライブラリは、こうした特徴を持つゲームを開発するために作られた、ということをもまずは念頭に置いていただきたい。

1.2 複数機種対応ライブラリという道具

今更言うまでもないことだが、複数機種で同時に発売する場合、手間をできるだけ削減するためには機種に依存しない部分と機種に依存した部分を分けるのが効果的である。そうすれば、機種に依存しない部分は全機種で使いまわせる。

例えば、ボールを打った時にどう跳ね返るのかは機種に依存しない。タイトル画面でボタンを押した時に何が起るのかも機種に依存しない。再生される音楽や効果音も同じにできる。やり方によっては、選手を描画するのにどのような計算を行うのかも機種に依存させずに作ることができる。このように、依存しない部分を増やすほど必要な労力が削減され、空いた力をより価値のある要素につぎ込むことができる。例えばゲームバランスの調整や、バグの撲滅に使うことができる。理論上はどれが一番開発が楽な機種でアプリケーションを開発すれば、勝手に他の機種でも動くようになるわけだ。

さて、一つのゲームだけを考えるならばアプリケーションの内部を機種依存部と機種非依存部に分けて開発しても良い。しかし、もし同じように複数機種対応するゲームが他にもあるのであれば、機種別に必要になる部分をあまりゲームに依存しないように作り、複数のゲームから使いまわせるようにできるとなお効果的だろう。さらに、機種に依存した部分を作るのに必要なスキルは、ゲーム本体を作るのに必要なスキルとはいささか異なる。このため、別の種類の人材を当てる方が効率が良い事が多い。その意味でも、機種依存部と機種非依存部をできるだけはっきりと分けることに利点がある。今回お話しする複数機種対応ライブラリの開発は、そのための手段である。

1.3 今回開発したライブラリの概要

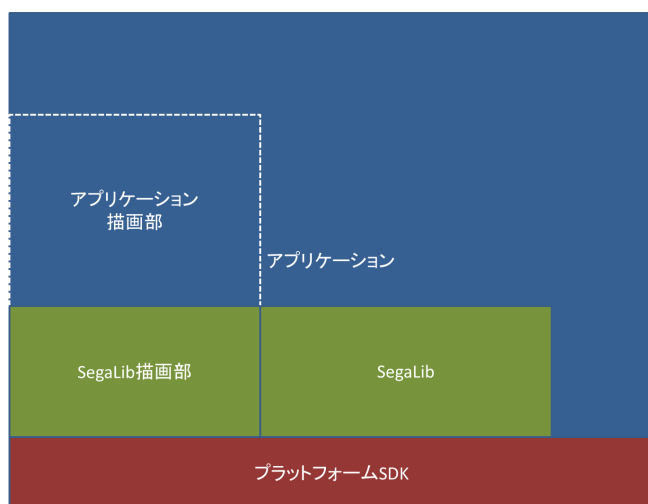
詳細について話す前に、今回作ったライブラリに何ができるのかだけを列挙しておく。

- 描画 API の抽象化
- OS システムコールの抽象化
- 音声再生・動画再生の抽象化
- アニメーション再生の基本部分
- 数学関数、ベクタ、行列、クォータニオン、乱数クラス
- STL 的なコンテナクラス群
- デバッグ文字描画、デバッガへの文字列ストリーム、メモリデバッグ支援

- 入力デバイス抽象化
- ファイル IO 抽象化
- XML 風テキストファイルの扱い

ライブラリがあることで、以上の機能を機種別コードを書かずにやれるようになるということだ。

以下は、SegaLib とアプリケーション、プラットフォームごとの SDK の関係を示すお決まりの図である。



アプリケーションとプラットフォーム SDK の間に入るものとして SegaLib が存在している。設計の話をする時にこの詳細を見ていくことにしよう。

ライブラリの名前

今回開発したライブラリの名前は「SegaLib」である。名前空間は Sega だ。正直大きく出すぎたかと思っているが、後悔はしていない。

後で述べるように、このライブラリは事実上 PowerSmash4 のためだけに開発され、実際それ以外にこれを使った作品は開発されていない。そしてセガ内で広く使われているわけではなく、そうなる見込みも今のところはない。このことは最初に断っておく。

第2章

開発時の状況

開発する状況は、設計や実装、運用のあらゆる面に大きな影響を与える。純粋に技術的な話だけをするわけには行かない。ここでは、開発がどのような状況で行われたかを述べる。

2.1 何故今更作っているのか

さて、複数機種対応ライブラリが手間を削減する上で有効であることにはあまり異論もない。しかし、2009年にもなると何故そんなものを作っているのか、という疑問は当然持ち上がってくることだろう。私も当時疑問に思ったし、今でもなお疑問に思う。そして、作らずに済ます方法が絶対になかったというわけではない。しかし、やってしまったものは仕方ないし、それなりには理由があった。

2009年当時、手が届く所に条件を満たすライブラリはなかった。条件とは、アプリケーションのコードとデザイン素材をひと通り作ればそれほどの手間なくPS3と360とWiiとPCの4機種で動いてくれる、という条件である。当時手が届く所にあったライブラリはアーケードとPCの2機種しか対応していなかった。PowerSmash3を作った当時、このライブラリはPS3にも対応していたのだが、その後のアップデートによってPS3のサポートが外されている。再度対応するにはかなりの時間が必要だろうと思われた。むろん、アップデート前のバージョンを掘り返してくるのも論外であった。担当者が昔のことなんて忘れてしまっているし、プラットフォーム側のSDKのバージョンも異なるのでそのまま動くわけではなかったからである。しかも、それに360やWiiを足すのは更に難しかった。

とりわけ問題になるのがWiiだ。WiiはPS3や360とは設計の世代が異なる。なんといってもシェーダが使えない。また、計算能力とメモリの量で大きく劣る。アーケードやPC向けに作られていたそのライブラリはギガ量のメモリがあることを前提に作られており、PS3ですらかなりの無理をして動かしていた。Wiiを相手にしようとするれば、もはや無理などと言えるレベルですらない。よしんばやり遂げたとしても、共通化オーバーヘッドと過去のしがらみの

ために使い物にならない性能になっていただろう。

あるいは、家庭用ゲームの部署にはそうした条件を満たすものがあつたのかもしれない。何しろ大企業であり、それら全てのゲーム機でゲームを出した経験を持っているのだから、何も無いということはないだろう。しかし、家庭用の部署は手が届く場所ではなかつたし、一つのゲームを全ての機械で発売したという話は聞いていない。発売まで至つた実績がなければ、仮にライブラリがあつたとしても信用には値しないし、そもそも手の届くところのないものを無理に使えば、サポートその他で大きなハンデを背負つて開発することになる。

さらに、最初はその4機種だけでいいとしてもいずれ新機種が現れるのは確実であり、そうなれば移植や続編の話が持ち上がることになるのもまた確実である。近年存在感を増している携帯電話の件もある。そう考えると、手元で対応機種を増やせる状況にしておくことは重要であろう。他人が作ったライブラリを使えば、対応機種を増やすことも他人に任せることになり、他人の都合で待たされたり諦めさせられたりすることになる。そのようなリスクは背負いたくない。実際、現在新しい機種への対応を行っている最中であり、この見方の正しさはすでに証明されていると言える。

以上の理由から、私は一から作ることが最適であると判断し、提案してみることにした。幸いライブラリの類は2回ほど作ったことがあり、過去の経験を活かすこともできるし、PowerSmash3の時にライブラリ使用者として経験したことも活かすことができる。私がやるのが適任だろう。いろいろあつたが、結果としてはその提案が通り、いろいろあつたもののゲームを発売まで持つていくことが出来たわけである。

なお、普通は組織において新しく作るという選択は嫌われる。既存のものがあればなおさらだ。実際、すんなりと決まつたわけではない。にもかかわらず何故通つたかと言えば、第一には当時の私に仕事になかつたからであろう。丁度本を書き上げた直後で、仕事らしい仕事になかつた。一応所属はライブラリ開発チームであつたが、既存のライブラリの保守業務を請け負っているわけではなく、言うならば宙に浮いた状態だつた。もし、既存ライブラリの保守に手を染めていればそんな提案が通つたかどうかは甚だ疑問である。

2.2 使えた資源

ライブラリの開発に使うことが出来た資源について述べる。これは設計を制約する条件として重要だからだ。時間と人材の二つである。

2.2.1 時間

PowerSmash4の発売日が2011年春であることは最初から決まっていた。開発開始が2009年初めなので、およそ2年だが、2010年冬にはゲームが完成していなければ危険であること

を鑑みれば、1年半とちょっと、ということになる。

もちろん、ライブラリができない限りアプリケーション側は本格的な開発には入れないので、ライブラリは可能な限り早く完成せねばならない。まずはPCで開発し、ある程度できてから他機種へ移植するのが開発効率も良く楽なので、まずはPC版のライブラリのみを作ってしまうこととした。それまでアプリケーション側では3のコードを眺めたり改造したりして時間を潰してもらわなければならない、この時間は極力短くせねばならない。

結果的には、PC版の初回リリースが2009年3月ごろで、その後、Wii、PS3、360の順に開発を行って順次リリースした。全機種で一応動作できたのが5月末である。ライブラリのおおよその形を示すのにおよそ4ヶ月程度かかったことになる。なお、この順序は制約の厳しい順である。先にゆるいハードウェアで慣れてしまうと後に制約が厳しい機種に移植した時に動かない可能性が高い。そこで一番厳しいWii版を先に作り、Wii版が動作する状態を保つように働きかけた。PowerSmash4の開発においてはWii版以外でメモリが足りなくなるケースはほとんどなかったわけで、これは成功だったと言える。

2.2.2 開発に使える人材

使える人間は私と、当時同じ部署にいた二人である。私は他の仕事を持っておらずPowerSmash4専任であったが、他の二人はアーケード向けライブラリの保守も担当しており、それほど拘束はできない。したがって、ライブラリ設計は小さくせざるを得ない。大きなライブラリを作ろうとすれば、間に合わないからである。

分担だが、開発環境構築と新しいSDKの習得に強いプログラマに各機種の環境構築から、入力デバイスのデータを取って簡単な絵を出すまでをお願いし、デザイナー向けツールからのデータの出力と変換の経験が豊富な人にいわゆるエクスポートとコンバータをお願いした。それ以外は私である。

なお、私についてもずっとライブラリだけ作ってれば良いわけではなかった。2009年7月には元いたライブラリ部署から、PowerSmash4のチームに異動になり、ライブラリが落ち着いた後はアプリケーション側のコードも書くことも期待されることになった。例えばプレイや選手アップ画面のカメラワークは私である。

このことはライブラリ設計にも影響する。もちろんライブラリにバグがあれば最優先で修正せねばならないので、アプリ側のコードを書くからといってライブラリの仕事をあきらめるわけではない。しかし、アプリ側のコードを書けば、アプリ側のコードの保守や改良の義務が生じる。自然、ライブラリの保守に使える時間は削られることになる。私がアプリコードを書き始めた後でもライブラリの保守に大きな手間がかかるような状態になれば、かなり危険なことになるだろう。保守コストの削減は極めて重要であった。実際、かなり末期までカメラワークの改善や修正の依頼が舞い込み、かなりの時間をそこに費やしている。なお、この異動

によってライブラリの使用者が隣の机にいることになり、これもライブラリ開発のスタンスに影響を与えた。これについてはのちほど述べる。

第3章

設計

ライブラリの設計は、さまざまな要因で決まる。理想のライブラリなるものは存在しない。性能、自由度、保守性、導入の簡単さ、バグのなさ、といったライブラリの品質に関する性質だけでも様々なバランスがありうるし、開発に使える人数、期限、といった作る際の制約もあって、何もかもを自由に決められるわけではない。また、使う側のスキルレベルも設計に大きく影響を与える。

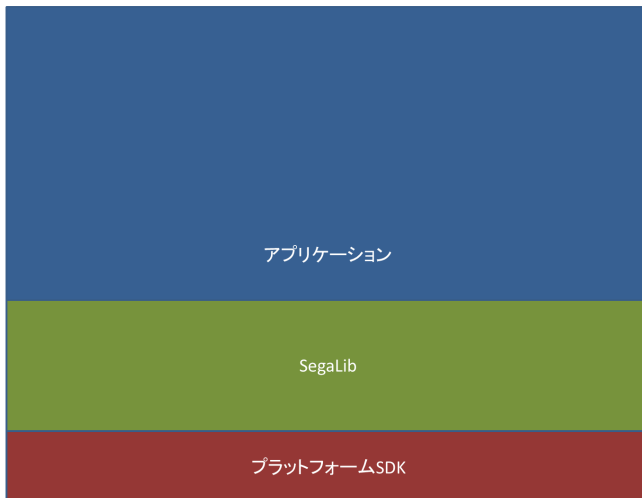
したがって、SegaLib は最初から汎用たることを目指していない。SegaLib が汎用になるとすれば、それは PowerSmash4 が発売された後で、他のチームが使いたいと言ってきた時である。その時に改めて汎用にするには何が必要かを考えて設計を考えなおせばいい、という考えだった。幸か不幸か、今なおその時は来ていない。

3.1 SegaLib の目的

ライブラリをわざわざ作ろうというのだから、当然目的がある。当たり前なことだが、しばしば忘れがちである。設計は何を目的として作るのかを決めた後の話だ。

結論から言えば、SegaLib の最大の目的はアプリケーション側の作業から、機種別の作業を可能な限り減らすことにある。自明のように思えるかもしれないが、全ての複数機種対応ライブラリがそうとは限らない。例えば、グラフィックス部分が機種別にバラバラで、関数名の末尾に Wii やら 360 やらがついているようなライブラリだってありうるし、実際見たこともある。目的が「全機種で最高の性能を発揮させること」にあるならば、このような設計が最良である。

しかし、SegaLib はこのような道を選ばなかった。多少性能を犠牲にしたとしても、共通コードで動くようにすることを目的とする。すでに述べたように、未来に機種が増えた時のことも考え、機種依存コードを最少にすることを最優先とした。



◆◇◆◆◆

これが理想の形だ。青いアプリケーションから赤いプラットフォーム SDK に触る手段がない、という状態が理想である。むろんのこと、実際にはこの理想は満たされない。それでも、青と赤の接触をできるだけ小さくする、ということが最優先課題となる。

3.2 ライブラリの範囲

SegaLib の目的はすでに述べた通り、「アプリケーションに機種固有コードを書かせないこと」である。そして、それを達成できる範囲で最も小さなライブラリにするという目標も抱えている。しかし、現実にはこれを完全に達成できるわけではない。最初からどの程度達成できるのかを見積もり、SegaLib の範囲というものをおおまかに定めておく必要がある。

3.2.1 含めないもの

例えば、360 には Kinect があるが、これは 360 にしかないものであり、どうラッピングしたところでアプリケーションには必ず機種固有のコードが書かれる。したがって SegaLib に用意することには意味がない。意味がないどころが有害ですらある。というのは、Kinect を最も理解しているのはアプリケーション側の Kinect 使用者であって、少々かじった程度で関数をラッピングしようとしても、最適なインターフェイスを提供することなどできず、単に SDK 側の更新時の手間を増やし、アプリケーションから最新の SDK にアクセスするまでの遅延を大きくするだけになるからである。すなわち、「他の機種でもできるがやり方が異なるもの」が SegaLib の守備範囲であり、「ある機種でしかできないもの」は SegaLib の守備範囲

ではない。

また、全機種に存在していてもインターフェイスの共通化が困難である場合もある。例えばセーブデータの扱いはダイアログ表示が必要だったりして機種ごとにより異なる。このようなものの抽象化は不可能ではないにしても、労力を必要とし、その割に得るものは少ない。

もう一つ、単純に間に合わなかっただけのケースもある。ネットワークはおそらく共通化が可能な部分もあるはずだが、私にネットワークの知識がなく、時間も足りなかったため、守備範囲外となった。もし私にネットワークの知識と、それを使って実際にゲームを作った経験があれば抽象化は可能だっただろう。しかし、そういった知識や経験を積むには時間が足りなすぎた。

3.2.2 含む必然性が薄いが含まれているもの

「なければアプリケーション側に機種固有コードが必要になるような機能」以外のものを一切入れずに SegaLib を作ることは不可能ではない。しかし、実際にそうすることは困難であり、有害でもある。

まず一つは、「機種固有の実装をしないと性能が大きく落ちるケース」である。例えば CPU でのスキニングがある。スキニングの計算は普通に C++ コードで書けばでき、必ずしも機種固有コードを書く必要はない。しかし、それでは実用にならない性能のものにしかならないわけで、CPU でスキニングをすることが必要なのであれば、SIMD 命令なりアセンブラなりを使った機種固有コードは書かざるを得ない。このような場合は SegaLib が面倒を見ている。スキニングの場合、インターフェイスは機種共通だが、実装は機種によって変えている。例えば PS3 では SPU を用い、Wii では CPU の SIMD 命令を用いる。

また、ライブラリとアプリケーションのやり取りのためにあった方が自然なクラスがいくつかある。例えば、ライブラリに 3D ベクトルを渡す場合、3D ベクトルクラスで渡すことが自然である。const float* で受け取るのは不自然であろう。

そして、実は一番数が多いのだが、アプリケーション側の作り方を規定するためにライブラリ側で持っているクラスというものがある。数学系のクラスのインターフェイスは、アプリケーションが遅いコードを書きにくくなるように設計した。XML パーサを含めることによって、アプリケーション側が独自テキストフォーマットをでっち上げることを防いだ。C++ 標準ライブラリの使用をできるだけ抑えてもらうために、性能的に危険な関数を除いた独自のコンテナ群を用意した。アニメーションやビットマップフォントなどの機能もこれにあたる。SegaLib の基本方針に照らせば、これらは必要とは言えない。むしろ外に出して追加ライブラリとして扱うべき機能である。しかし、これらを追加ライブラリにせず中に持つことによって、余分な手間なくアプリケーション側から使用できるようにし、アプリケーション側に望ましくないコードの書き方をしないように提案している。

もっとも、単にアプリケーションのコードに追加するのが面倒で、つい普段作業しているライブラリのコードに追加してしまった、という側面もあり、後で外に出すことになる可能性もある。実際、交差判定用のライブラリも作ってはみたのだが、SegaLib 使用者全員が使うという代物でもないので、外に出しておいた。

3.3 関数の集まりか、それともそれ以上のものか

ライブラリには単に便利な関数の集まりにすぎないものから、アプリケーションの作り方を積極的に規定するものまで様々なものがある。C/C++ 標準ライブラリは前者であろうし、DXUT のように実装する関数の名前を指定されたり、コールバックを登録したりするタイプのライブラリは後者と呼べるだろう。

SegaLib は後者である。

アプリケーションの作り方をかなり強く規定する。例えば、ある部分だけ使う、ということはいけない。一つのパッケージとして提供しており、SegaLib を使うなら、グラフィックスも、音も、ファイル IO も SegaLib を使わなくてはならない。所定の手続きに従って初期化すれば、それらのものが動く状態になる。カスタマイズは許さない。どうしてもしたければ、ソースコードを書き換えてもらうことになる。その時点で私はサポートをしない。

また、エントリポイントはライブラリ内部にあり、アプリケーション側が書くことは想定していない。

```
#include "Sega/Window/Window.h"
namespace Sega{
    Window::configure( Configuration* ){
    Window::update(){
}
}
```

これは SegaLib を使った最小のアプリケーションのコードであり、これだけ書いてリンクすれば全機種で実行できるようにしてある。エントリポイントは機種依存であり、SegaLib の目的に照らせば当然ライブラリ内部で面倒を見るべきである。当然の帰結として、このように「決まった名前の関数を実装してもらうと、然るべきタイミングで呼ばれる」という作りにならざるを得ない。このように強い制約を課している段階で、単なる関数の集まりとは言えない。

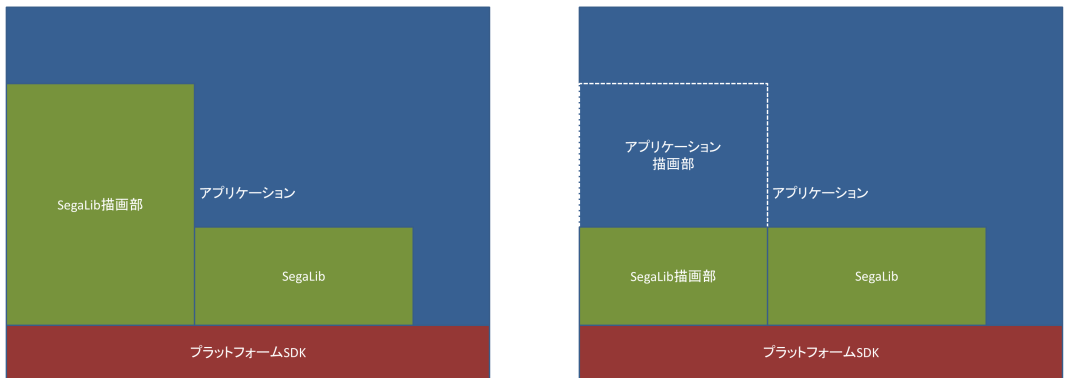
確かに、カスタマイズしたいという要望はありうるだろう。例えば「ファイル IO は独自でやりたい」というような場合もあるだろうし、グラフィックスはギリギリを目指すので SDK を叩かせる、という場合もあるだろう。しかし、さしあたって PowerSmash4 においてはそんな要望はありえない。PowerSmash4 を作るために作っているライブラリなのだから当たり前である。将来そういうチームが現れた時に考えれば良いだろう。機能追加で対応するのか、あるいは単にソースコードごと渡して勝手にやれと言うのかは、その時に考えれば良い。そんな

未来のことを考えても無駄である。

3.3.1 描画エンジン？

描画エンジンという言葉をよく聞く。定義は定かではないが、おそらく「drawModel() と書くと画面に絵が出る」という程度に上位層までを含む描画用のライブラリのことであろう。

その意味で言うならば、SegaLib は描画エンジンではない。



もし、描画エンジンなのであれば、構成図は左の図のようになるはずだが、現実には右の図の通りである。

グラフィックスのインターフェイスは DirectX や OpenGL に近い。シェーダはアプリケーション側で書き、モデルのデータフォーマットすら規定しない。それは SegaLib の外とし、サンプル実装のみを同僚に頼んで用意してもらった。なるほど、enableBloom() と書けばブルームエフェクトが有効になるようなライブラリは便利ではあろう。しかし、PowerSmash4 はチーム内で絵作りを設計し、チーム内でシェーダを書く前提のチームである。そのようなチームに対してそんなものを供給しても邪魔になる。また、品質と性能のバランスを調整するのもアプリケーションチームの役目であり、SegaLib はそこを邪魔してはならない。それに、そもそもそこまで実装するような余裕はなく、保守する余裕はさらさない。結果、アプリケーション側に描画のためのかなり大きなコードがあって、初めて絵が出る構成となっている。

いずれ、「自分でシェーダなんて書きたくない！」というチームが SegaLib を使いたいという話になれば、この「アプリケーション描画部」に相当する部分をこちらで用意することもあるかもしれない。しかしその場合も、それは SegaLib の外に置かれることになる。絵の好みやゲームの性質によって大きく設計を左右される部分であり、設計変更は頻繁になるし、陳腐化も早かる。そのようなものを SegaLib の中に入れて全体の保守性を落とすのは望ましくない。

3.4 品質のバランス

品質は高いほど良い。しかし、品質といってもいろいろある。性能、自由度、保守性、導入しやすさ、学びやすさ、バグのなさ、デバグの容易さ、コンパイル速度、などの全てが品質だ。これをどうバランスするかはライブラリを設計する上で非常に大切である。このことを考えておかないと、全体の品質に致命的な悪影響を与えることになる。例えばコンパイル速度は非常に地味でしばしば軽視されるが、これがあまりに遅いと開発工程のあらゆる所に悪影響を及ぼす。デバグや導入の容易さも同様である。

3.4.1 性能

性能は常に高いほど良い。しかし、複数機種を対象とする段階で、性能は最優先課題ではありえない。複数機種で発売されるゲームは、基本的に単一機種で発売されるゲームほど機械の性能を引き出せない。これは自明である。

しかし、PowerSmash4においては綺麗な絵が望まれており、性能、とりわけグラフィックス性能を無視はできない。無視できないどころか、目標をかなり高い所に置かなくてはならない。しかし、あくまで SegaLib の目的は機種固有コードをアプリケーションに書かせないことであり、まずはそれを最優先とせねばならない。もちろん、あまりに性能を落とすような設計は受け入れられないわけで、場合によっては機種固有コードを残さざるをえないこともあるだろう。しかし、どの程度性能にダメージがある場合にそうするか、ということは常に考えねばならない。

SegaLib における基準は二つある。一つは「性能制約が厳しいハードウェアほど例外を許す」ということ、もう一つは「どれかの機種が不利にならざるを得ない場合は、性能が高い機種に不利を押し付ける」ということである。具体的に言えば、Wii は多少の機種固有コードを書かせることもやむをえないし、共通化できる場合には多少 360 で遅くなっても Wii に合わせる、ということである。こうすることで、機種固有コードを最小にしつつ全機種で同じものを動かす、ということを達成しやすくなる。

また、Wii が存在すること、そして、今後携帯機や電話も相手にすることを鑑み、速度よりも容量を重視することを基本設計とした。速度は遅くても処理落ちするだけである。しかし、メモリが足りなければそもそも売ることができない。どちらが重要かは言うまでもない。それに、メモリ帯域にかかる負担が減ることで、かえって高速化する可能性もある。メモリ帯域がますます高くつく方向に進化している現状を鑑みるに、こうなる可能性は高い。容量は増えているが、速度はそれほど上がっていないのである。

3.4.2 自由度

自由度が高いというのは、やろうと思った処理ができることである。SegaLib は、アプリケーションがやろうと思うことはできるようにせねばならない。そして、SegaLib の目的から考えて、それはアプリケーション側が勝手に SDK を叩くのではなく、SegaLib を通して行うのが理想である。したがって、SegaLib はアプリケーションが必要とする SDK の機能を全てラッピングすることが望ましい。しかし、すでに述べたように人材的にも時間的にも制約があり、また、機種依存の機能などはラッピングしない方が良いこともある。

SegaLib においては、PowerSmash4 が必要としない機能は一切入れていない。入れてもすぐには使わないし、テストもされない機能は入れるだけ無駄である。たとえば、PowerSmash4 の描画機構は、Z バッファをシャドウマップとしてそのまま使う手法を採用していない。そのため、現在の SegaLib は Z バッファをシャドウマップに使う機能を提供していない。こういったことが無数にある。その意味では自由度は高くない。他のアプリケーションの開発に使われるようになれば、おそらく大量の機能追加が必要になるだろう。

また、自由度を広く解釈すれば、「やろうと思った処理を楽にできること」とも言える。「いろんな機能があるほど自由度が高い」と考えている人は多いはずだ。アプリ側で書けばできるがライブラリが持っている方が便利な機能、というのは確かにあり、こういうものが多いライブラリを「自由度が高い」と言うことはある。しかし、資源の制約から SegaLib ではこういったものを極力入れないようにしている。欲しければアプリケーションで書け、というスタンスだ。アプリケーションの中に私が書いたり、SegaLib の周辺ライブラリとして提供した機能はあるが、SegaLib の中にはできるだけ入れないようにした。その意味でも自由度は低いライブラリである。

3.4.3 保守性、学びやすさ

保守性はコードの量におおむね反比例する。学びやすさはインターフェイスの大きさにおおむね反比例する。ゆえに、SegaLib はコードの量とインターフェイスの大きさの両方を最小にすることを目標とした。

もちろん、時間的、人的な制約から言って、そうせざるをえないし、機能が少なければ自然とそうなる。しかし私は保守性の悪いライブラリと学びにくいライブラリを憎んでおり、時間や人に余裕があったとしてもそれらは重視したであろう。

まず保守性に関しては、すでに述べたように機能を最小限にすることが重要である。「あれば便利」という程度の機能は SegaLib の中には入れない。また、使っていない機能を積極的に消すことも含む。

学びやすさは、インターフェイスの大きさだけでなく、一貫性によっても変わってくる。あたかも同じ人間が全てのクラスのヘッダを書いたかのように見えることが望ましい。そのため、コーディングのスタイルは厳密に統一した。最初手伝ってもらった場所に関してはスタイルが若干異なっている部分もあったが、あとから全て改めた。自己中心的であるかもしれないし、個性を認めない考え方とも言えるが、使用者にとってはコードを書く人間の自己主張は無用であろう。

また、使用者にとっては無用である `private` 部は極力小さくし、継承の使用も最少に抑えた。別のヘッダを見て複数のクラスの定義を眺めなければ全貌が見えないという意味で、継承は学びやすさを阻害するからである。むろん、使用者が直接使うことがないクラスのヘッダは隔離し、`include` 以下には使用者が直接使いうるクラスのヘッダのみを配置するようにした。

3.4.4 導入のしやすさ

SegaLib を使ったアプリケーションの開発は極力楽でなくてはならない。そのために、SDK 側のライブラリで必要なものは SegaLib のライブラリファイル内に含めてしまい、独自に特殊な機能を使わない限りは `libSega.a` や `Sega.lib` のみをリンクすれば良いようにした。

また、直接の関係はないが、各機種向けの SDK もこちらでパッケージ化して、配布している。バラバラに各プラットフォームの開発サイトにアクセスしてダウンロードする必要はない。

もう一つ重要なのは、ヘッダのインクルードに一つのルールを設けることである。例えば Matrix34 クラスを使いたいと思った時に、何をインクルードしなくてはいけなにかは使い勝手に影響を与える。SegaLib においては、Matrix34 クラスを使いたければ `Matrix34.h` をインクルードすれば良い。そのクラスのヘッダをインクルードすれば良く、それ以外には必要ない。全てのヘッダファイルについて、インクルードする 1 行以外に何も無い `cpp` がコンパイル可能でなければならない、というのが SegaLib の掟である。例えば、

```
#include "Sega/Graphics/VertexBuffer.h"
```

とだけ書いた `cpp` はコンパイルを通る。あらゆるヘッダについてこれを保証する。もしコンパイルエラーになればそれはバグなので、必ずこちらで修正する。実際にどうやってこれを実現するかは場合により、`class Vector3;` と書くだけで済むならそうするし、さらなるインクルードが必要ならば、`Matrix34.h` の先頭において `Vector3.h` をインクルードする。

「使いたいクラスのヘッダをインクルードすれば使える」というのはいかにも当たり前のことに思えるかもしれないが、これを満たさない作りのライブラリは多い。例えば Matrix34 が `Vector3` を引数として取る関数を持っているために、先に `Vector3.h` をインクルードしておかねばならない、というようなライブラリは良く見かける。また、クラス名とファイル名が一致

しないケースが多いと、そのクラスを使うのに何をインクルードすればいいのかを調べるのが面倒である。SegaLib においてはそのような例は極力少なくしておいた。

ただし、利便性を高めたいからと言って「Sega.h をインクルードすれば全機能が使える」というようなことはしない。これをすれば不必要なヘッダを大量にインクルードすることになってビルド時間が大幅に増すからである。コンパイル時間の重要性は軽視されすぎている。常に並列ビルドツールが使えるとは限らない、ということは強調しておきたい。ただし、もしこのようにしてもアプリケーションのビルド時間が顕著に伸びないようにできるならば、こうした方が良いだろう。あるいは、アプリケーションのビルド時間が十分に短いならば、それによって多少伸びたところで問題はない。かなりの大きさを持ったアプリケーションであっても 1 分かからずにビルドできるのであれば、Sega.h で全てを済ませるやり方に移っても良いように思う。

3.4.5 バグのなさ、デバグの支援

バグのなさには二つある。一つはライブラリのバグのなさであり、もう一つはライブラリを使うアプリケーションのバグのなさである。ライブラリの使い方がわかりにくいことはアプリケーションのバグを増やすため、後者に関してもライブラリはかなりの部分責任を持つ。

アプリケーションのバグを予防するには、ライブラリの使い方を極力単純化するのが良い。例えば二つの関数があって、その効果が独立して見えるものなのであれば、どの順番で呼んでも良いようにすべきである。また、ある二つの関数が常に二つセットで呼ばれるのであれば、そもそも一つに統合する方が良い。そういったインターフェイスの設計がもっとも重要である。それで防ぎ切れないケースでは、間違っただけで使った時に即座に `assert` に引っかかって停止するようにすることが有効になる。性能に多大な悪影響を及ぼさない限り、引数や呼び出しタイミングに関する `assert` はできるだけたくさん仕込むべきである。

一方、ライブラリの中のバグに関しては少々犠牲にした。一応各機能のテストはあるのだが、あまり周到なテストではない。例えば、テストが自動で走って通らないとコミットできない、というような先進的な開発体制は敷いていない。もちろんバグは発売前には全て直すのが、途中のリリースにおいてバグがないことは保証していない、ということである。なんと言っても、最初の数カ月が過ぎた後は開発は私一人であり、しかも主な使用者が隣に座っている。隣の彼には迷惑をかけたが、要望に素早く応えることを優先し、テストは軽視した。しかし、私の記憶の限りにおいてではあるが、それでそれほど大きな問題が出たことはないように思う。

もう一つ、デバグ支援についてだが、この優先度はさして高くない。そもそもバグを入れにくい作り方を強要する方が費用対効果は良く、発生してしまったバグに対してできることはあまりない。

基本は「性能に大きな害を与えず、実装が大きく複雑にならない範囲で、支援機能を入れる」

である。つまり、性能と保守性が上位に来る。これは、過去のライブラリがデバグ支援を手厚くしすぎて性能を大きく損なっていたことへの反省である。当然手厚い支援は大きな実装を必要とし、保守性も悪くする。デバグ用の文字列出力、画面への文字出力、assert の整備、そして、メモリリーク検出、といった基本的なものにとどめている。

3.5 更新と作業工程

SegaLib はアプリケーションと並行に開発しているライブラリである。また、すでに述べた通り、とりあえず PC 版だけ作った段階で渡してしまい、他の機種については後で考えるというやり方となった。その PC 版にしても、最初は音声再生や非同期ファイル IO は後回しである。したがって、機能追加と同時に元の設計ではまずい所が必ず出てくる。とりわけグラフィックスについてはこの変更が激しい。機種によって違いが大きく、かつ規模も大きいのがグラフィックスだからだ。

例えば、シェーダ定数についての設計は二転三転している。最初は DirectX9 のようにレジスタ番号でシェーダ定数をセットするしくみにしたが、後々 OpenGL 的にレジスタの概念がない SDK が出てきて根本的な変更を迫られた。どうにか DirectX9 風のままで行こうと小細工を繰り返した時期もあったが、結局最初から再設計することになった。クラスの数が増減するレベルの変更である。なるほど、こうした変更が生じるのは元の設計が考えぬかれていなかったからであり、弁解の余地はない。しかし考えるのに時間をかけて最初に動くまでに時間がかかりすぎれば、それはそれで迷惑をかける。

SegaLib は拙速を良しとし、更新をためらわないことを基本方針とした。

なるほどよほどの仕様変更でない限り、インターフェイスを変えずに内部で対応することは不可能ではない。しかし、そうやって出来たものが性能と保守性の両面で劣ることはほぼ間違いなく、長期的には災いの種となる。PowerSmash4 は移植等で長期化が予測されており、下手な妥協は PowerSmash4 にとってもためにならない。アプリケーション側にどんなに大規模な修正が必要になったとしても、より良い設計が思いつけば躊躇なく変更することとした。短期を犠牲にして、長期を取る、ということである。

そういうわけで、機種を追加したり、機能を追加したり、見落としに気づいたり、より良い設計が思い浮かんだりする度に、アプリケーション側では修正が強いられることとなった。場合によっては、テキストチャやアニメーションのバイナリデータフォーマットすら後から変えるケースがあり、その場合には全てのデータを変換し直す羽目になった。それも 1 度や 2 度ではない。古いインターフェイスも残す、という選択肢もあるが、ショー出展寸前でもない限りはそのようなことはしなかった。そうした暫定措置はしばしば永続化してしまうものであり、万が一いつまでも残ることになれば保守コストがかさむ結果になるからである。

もちろん最初からそういうことが起こるという前提でアプリケーション側には極力変更が強

い作りにもしてもらったし、そうするように働きかけもしたが、それでも負担であったことは間違いない。

3.5.1 更新時の連続性を重視しない理由

SegaLib は未来においてはともかく、現在は PowerSmash4 以外には使われていない。したがって、PowerSmash4 の開発に最も寄与するようにライブラリを開発すれば当面の使命は果たすことができる。これがもし、複数プロジェクトから使われていれば、変更点の通知と説明、トラブル発生時の対応などのコストが何倍にも膨れ上がり、変更を行うことによる利益が目減りすることになる。しかし、今回はそうではないし、そうならないことを確認した上で始めている。

新しく作るライブラリは最低一つのゲームが発売できるまでは複数プロジェクトを相手にすべきではない。これは非常に重要である。実際他のチームが使いたいと言ってきた時には、「私は一切サポートしない。好き勝手改造するし、それを通知もしない。それでもよければ勝手に使え」というおおよそ無責任な事を言い放った。実際そのチームはそんな状況でも自分で手を入れながら使っていたようであり、非常に申し訳なかったが、仕方ないものは仕方ない。そうしなければ SegaLib は窒息して死んでいたであろう。

加えて、私は PowerSmash4 チーム内の人間である。隣の席には SegaLib の変更によって最も大きな面倒をかける描画担当者が座っており、「ごめん、あそこインターフェイス変えるわ」と一声かけるだけで気兼ねなく SegaLib の変更を行うことができた。彼には大変な苦勞をかけたので申し訳なく思うが、当然彼の優秀さを見込んでのことである。もしその席に経験の浅い者が座っていればもう少し考慮しただろう。また、もし私がチームの外にいれば、コミュニケーションが困難になるため、更新の頻度や程度を緩めていたことと思う。

そして、PowerSmash4 が発売される頃には、変更のペースも落ちて落ち着くであろう、という見積もりは当然あった。そうでなければ発売できないのだからそうなるはずだし、そうする以外にない。一刻も早く落ち着かせるためには、変更を先延ばしにしてただでも忙しい開発末期の時間を奪うべきではなく、修正点が見つかったらすぐにでも修正を行う方が良い。変に躊躇して、結局発売直前に直さねばならないことが発覚する方がよほど恐怖である。

最後に、最初に述べた基本方針がある。「小さく保つ」ことは保守コストの面から絶対の条件であり、互換性のために過渡的なコードが長く居座ることはこの方針に反している。

余談:ライブラリの肥大化について

余談になるが、ライブラリはその性質上どうしても肥大化しがちである。とりわけ複数プロジェクトに使われれば肥大化は避けがたい。あるプロジェクトに要求されて足した機能が二度と使われない、ということが続けばどんどん肥大化する。不要になったものを積極的に捨てな

けれどもならないが、一回足してしまったものは誰が使っているかわからないのが普通であり、肥大化は避けられない運命にある。もちろん、PowerSmash4 しかない現在の SegaLib においても、試しに足してみたが結局使わなかった、というようなことがあればやはり肥大化する。ただしこの場合は使われていないことを確認するのは容易であり、折に触れて使われていない機能を探して削除すれば良い。実際 SegaLib ではそうしている。

将来 SegaLib が複数プロジェクトから使われるようになれば、この問題は深刻化するだろう。その場合にどうやって肥大化を抑えるかについてはまだはっきりした方針はない。そうならないとわからないというのが正直なところである。しかし、使っているプロジェクトがわかっている、そのソースコードが手に入るのであれば、「ある関数がどのプロジェクトから使われているか」を自動で調べることは容易である。その結果すでに終わったプロジェクトにしか使われていなければ削除して良いし、使われている機能であっても場合によっては削除はできる。プロジェクトに交渉しても良いし、その機能をライブラリ外で実装してそのプロジェクトに渡したり外部ライブラリ化しても良いだろう。とにかく、「作ったものを捨てる」という手間をかけることを忘れないようにすることで、肥大化のペースを抑えることはできる。それで抑えられなくなるほど肥大化したとすれば、それはライブラリに寿命が来たということであろう。バグ修正以外の変更を止め、新しいライブラリの制作に力を割くべきである。むしろ、そもそもライブラリを作る必要があるか、というところから問い直す必要があることは言うまでもない。

3.5.2 アプリケーション側の対策

アプリケーション側は、下にあるライブラリがこのようにコロコロと大改造を繰り返すことを見越しておく必要があった。一番大切なのは、最も変更が大きくなるグラフィクス部分は一人で担当し、カプセル化しておくことである。そうすれば、どのように変更されても、その人が対応させれば他の人の手間は無い。いわゆる「描画エンジン」的な部分をアプリケーション内に作っておいて、アプリケーションのほかの部分があるところを通してしか描画を行わないようになっていけば、SegaLib のグラフィクス部分がどのように変更されても、描画エンジン部だけをいじれば済む。

実際、SegaLib は配布の仕方からしてこのことを念頭に置いている。私がコミットしたものはグラフィクス担当の彼以外は持って行かない。つまり、私がコミットしたものは直接はアプリケーションからは参照されない。彼が持って行ってアプリのビルドが通って動作することを確認した後で、彼がアプリケーションから見える場所にビルド済みの SegaLib をコミットする。これではじめて他のアプリケーションプログラマに新しい SegaLib が渡ることになる。繰り返すが、その彼が私の隣に座っていることは極めて重要である。コミュニケーションの頻度は席の配置を決める際にもっとも重視されねばならない。

なお、この配布体制は場合によっては不便である。例えばファイル IO や音声に変更があった場合にはグラフィクス担当にとっては専門外となる。実際、サウンドに不具合があって更新とテストを繰り返す際には、例外的にサウンド担当に直接渡してビルドしてもらうこともある。だから、そうできるようにはしておくべきだ。しかし、ほとんどの変更はグラフィクスであるし、グラフィクス以外の機能はインターフェイスも簡素で、変更があったとしても大した規模ではない。よって、この問題はさほど重要ではない。高頻度で修正されるライブラリがいきなり全アプリケーションプログラマに配布されるような状況は悪夢以外の何者でもなく、仮に比較的安定したライブラリであったとしても、更新版をアプリケーションに組み込む担当者は一人であるべきである。ちなみに、PowerSmash3 の時には私がライブラリを受け取って描画エンジンを組みライブラリをチームに配布していた。

もう一つ、データの問題がある。モデルフォーマットを拡張したり、アニメーションの圧縮効率を上げたりした場合には、古いバイナリデータが読めなくなることがある。もちろん普通のライブラリであれば、こんな大変更はそうそうしない。多少の非効率があっても互換性を保つ範囲でしか変更せず、完全に新しいバージョンを作る時にのみそういう変更を行うのが普通だ。しかし、発展途上にあるライブラリではそんな悠長なことは言っていられない。そこで、ボタン一発で全データを新しいコンバータを通して新フォーマットに更新できるよう、バイナリ化前のデータをサーバに置いておくようにした。バイナリ化した後のデータだけがサーバに置かれ、その前のデータは各アーティストの PC にしかない、というような状況ではこれは無理である。

また、PowerSmash4 のゲームとしての規模がさほど大きくなく、全データをコンバートしなおしても一晩あれば十分終わる量であったことも重要である。PowerSmash4 のデータ量はたかだか 4GB 程度であり、サーバに必要とされる容量も高が知れている。もちろん、それでもデータフォーマットの変更は大変な手間とリスクを伴うため、SegaLib には慎重な手順を経て行われた。ほとんどデータが完成した発売 4 ヶ月前にアニメーションのフォーマットに大改造を加えた時にはさすがに響きを買ったが、性能上必要な改造だったのは事実であり、後悔すべきはあの時点で変更を行ったことでなく、その時点までその変更を思いつかなかったことであろう。

というわけで、ライブラリ担当者は多少の無茶をしても許されるキャラクターを浸透させておくべきである。「まあ平山のやることだし」と言ってあきらめてもらえるようになればしめたものであり、むろんのことそれを裏付けるだけの働きを日常からしておかねばならない。人間関係が円滑でなければ、正義は行えないということだ。むろん、私が円滑な人間関係を構築できているかどうかは、私にはわからない。チームの皆が私に恨みを抱いている可能性は十分にある。

第4章

実装

以下では、ライブラリの各部分についての設計と実装について、出せる範囲でできるだけ詳しく述べる。

4.1 ビルド構成

VisualStudio 用語で言うプロジェクトのディレクトリ構成や、ソースコードの置き方について述べよう。そんなことは枝葉末節だと思われるかもしれないが、ここを甘く見てはいけない。ここを間違えば、確実に保守コストと導入コストが増大する。

4.1.1 1 ライブラリ構成

モジュール化が流行りである。大きな物は複数の部品に分けて作り、分業しやすくするとともに、あわよくば使用者側での取捨選択を可能にする。

このこと自体に異論はない。取捨選択できないよりはできた方が良く、分業しにくいよりは分業しやすい方が良い。問題は、これをどういった手段で実現するか、そして、それが何を犠牲にしてなされるかである。

結論から言えば、SegaLib では.lib やら.a の類、つまりライブラリファイルは1つしか作らない。一つにグラフィクスや音声、パッド入力に数学まですべて入れてある。複数にするのは複数にした方が都合がいい事情があるからで、そういった事情がないならば複数あるよりはひとつだけの方が楽に決まっている。SegaLib の場合はそういう事情をすべてつぶしてあり、ひとつでも問題ない状態にした。

まず、モジュールごとにバラバラにリリースすることがありえない。グラフィクスライブラリと音声再生ライブラリが別々にリリースされるのであれば、分けねばならない。しかし、SegaLib においてそのような需要はほぼない。すでに述べたように SegaLib は小さく、シン

ポル情報を入れなければ 2MB とない。ビルドも高速で、全体をビルドしてもせいぜい 1 分、PC 版であれば 15 秒である。あまりにビルドが遅いのであれば、変更した部分だけをビルドして高速化したいと思うこともあるが、この程度であればまるごとビルドしてもかまわない。ライブラリファイルを分けるとプリコンパイルヘッダの生成がライブラリファイルごとになるためにビルド時間はかなり増えるし、そもそも機種ごとに 5 個も 6 個もライブラリのビルド設定をするのは面倒くさい。アプリケーション側にして一つで済むならその方が楽に決まっている。

また、ライブラリファイルを分けたからといって分業しやすくなるわけではない。ディレクトリ構造がモジュールの分割を反映していれば、それで分業に関する問題はほぼなくなる。そもそも現代には Subversion というツールがあり、仮に同じファイルを複数人で編集したとしてもさほどの問題は起きないのである。

例えば、PowerSmash3 や、その後私が関わった作品で使ったライブラリは 6 か 7 個のモジュールに別れており、それぞれが別のライブラリファイルを生成していた。拙著「ゲームプログラマになる前に覚えておきたい技術」においてもそうである。モジュールごとに vcproj を分けた理由は全く思い出せない。おそらく理由もないままに先入観でそうしてしまったのであろう。ソースを持ってきてビルドするには個別にビルドを走らせねばならず、面倒だし時間もかかった。また、モジュール間の依存関係のためにそれぞれのバージョンが不整合を起こす危険もあった。不整合があっても表面上動いてしまうケースが多く、問題の発見が遅れることもある。使う側からしても相当に面倒だったわけで、作る側にとってはなおさらそうだろう。それに、ああいうものは結局全体をひとつのパッケージとして提供せざるを得ず、理念の上では取捨選択が可能であっても、実際にそのようなことはまずなされないのである。

むろん、過去のライブラリがわざわざ分けていたのにも理由があり、ビルド時間が長くモジュール単位でのビルドが必要だったり、そもそも機能が非常に多く規模が大きいためにディレクトリを分けた程度では不十分だったりしたからなのだが、そのどちらの問題も SegaLib では前もって除いてある。

4.1.2 Debug ビルドと Release ビルド

SegaLib は Debug ビルドと Release ビルドの 2 つの構成を持つ。Debug ビルドでは全ての assert が有効になり、メモリ確保や解放の度に塗りつぶし処理が走る。Release ビルドは限られた数の assert のみが有効になり、メモリ確保や解放の際の塗りつぶし処理が行われない。最適化を除いた機能的な面では、この程度の差しかない。

SegaLib は Debug ビルドであってもあまり遅くなりすぎないことを念頭に置いている。Release ビルドに比べて 1/3 以下の性能になることは許容しない。Debug ビルドがあまりに遅すぎると、Debug ビルドでゲームを動かすことが苦痛になりすぎて結局使われなくなり、

Debug ビルドで動かせばすぐわかったことがいつまでもわからない、という状況に陥るからである。これは過去のプロジェクトで身を持って味わった。当時使っていたライブラリは STL 等で大量の小さなインライン関数を通る場所が多く、また、過剰なまでに `assert` が仕込まれていたために、Debug ビルドの実行速度があまりにも遅かったのである。SegaLib ではインライン関数が展開されなくてもさほど遅くならないように配慮すると同時に、`assert` の数も抑えておいた。どうしてもバグの発見を優先したい場合には別途マクロを定義するようにし、通常の Debug ビルドの速度は一定以上に保つよう配慮した。

逆に Release ビルドにもデバグ機能を極力残すようにした。完全にデバグ機能がない状態ではテストが出来ない。止まっても「止まった」ということしかわからない。これでは出荷の寸前まで有効なテストができない。そこで、デバグ機能は極力残すようにした。例えばメモリリークのデバグ機能は Release ビルドでも残っている。このため、Release ビルドと Debug ビルドでメモリの使用量が全然違うということはない。過去に使っていたライブラリでは、Debug ビルドであまりに大きなオーバーヘッドがあったために、Debug ビルドではメモリに入らないので Release ビルドしか使わない、というプロジェクトが現れ、デバグ作業に非常に困難をもたらした。もちろん Debug ビルドでも収まるようにしなかったプロジェクト側に問題があるのは明らかだが、そのように差があるライブラリを作ったことにも責任がないとは言えない。SegaLib ではこの経験を踏まえて、Debug ビルドと Release ビルドでメモリの使用量がなくなるようにしてある。

なお、`assert` の扱いについては後で詳しく述べる。

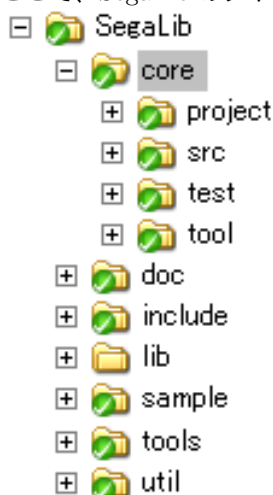
4.1.3 cpp と h

通常 C++ においては実装を `cpp` に書き、インターフェイスを `h` に書く。これが基本の作法である。

しかし、「そういうものだ」と思う前に、一回くらいはなぜかと考えてみるべきである。`h` に実装を書かないのはなぜかと言えば、`h` が説明書としての役割を果たすからである。使用者にとってみれば、`h` で紹介されているクラスなり関数なりの使い方と機能がわかれば良く、それをいかに実現しているかは無用な情報である。この無用な情報は `cpp` に隔離すれば良い。すなわち、コンパイラの事情を無視すれば、`cpp` なるものはまずは隠蔽のために存在する。

もちろんコンパイラの事情も無視できないが、そもそも C++ 言語そのものには `h` や `cpp` という区別はなく、`#include` で結合された後のファイルがコンパイラにとってはすべてである。`h` や `cpp` の区別は人間の都合により、その都合とはインターフェイスと実装を分離したいという都合である。分割コンパイルは一部を書き換えた時に全体がコンパイルされて時間がかかるのを防ぐためにあるが、この効果も後述するように今となってはさほどではない。すなわち、コンパイラの都合から言っても、`h` と `cpp` を分離することにはさほどの意味はない。

ここで、SegaLib のディレクトリ構成を図で示す。



SegaLib においては、include 以下に使用者から見える h ファイルが置かれている。ここに
ある h ファイルには使用者が知る必要のない情報は極力含めていない。インライン化やテンプレートのためにやむを得ず実装を h に書かねばならない場合も、別ファイルに分けて#include
することで極力目ざわりにならないようにしている。例えば Math.h であれば、MathImpl.h
といった具合である。そして、それ以外の実装はすべて core/src 以下にあり、これは配布され
ない。

なお、実装という言葉の範囲は誰の立場に立つかによって異なることに注意されたい。使用
者の立場からは、使用者が知る必要がないものは全て実装である。内部でしか使っていないク
ラスは h に書こうが cpp に書こうが実装である。そして、h ファイルだからといって include
以下に置かねばならないという決まりはない。実装であるがゆえに、構造はどのようなもので
あっても使用者の使い勝手は悪化しないのである。

SegaLib においては、内部クラスのほとんどは h と cpp にはわかれていない。可能な限り
すべて h で実装を行っている。JAVA や C# の経験があるならそれを思い起こせばいい。グ
ローバルあるいはクラススタティック変数の実体を定義する時と、内部クラス間で相互参照が
ある場合にはコンパイラの都合上 cpp を設けざるを得ないが、そうでなければすべてを h に
書いても不都合はない。むしろその方が全体の記述量が減って見通しが良くなる。

そもそも、cpp は少ないほどビルドが短い。コンパイルのかなりの時間は#include のため
のファイル IO が占めている。システムや標準ライブラリの h ファイルが非常に肥大化した結
果、たいていの cpp では膨大な数のシステム系 h ファイルのインクルードを強いられている
からだ。自作の h ファイルの占める比率はわずかであり、例えば Windows 上の開発において
windows.h をインクルードすれば、その先には数十万行のコードが待っている。これが cpp
の数だけ行われるのがコンパイルの遅さの原因である。コンパイル処理そのものにかかる時間

は CPU の高速化によってどんどん短くなっているが、ファイル IO に要する時間はあまり変わっていないため、時間が経てば経つほどインクルードの処理時間の比率が増している。

プリコンパイルヘッダは大きな助けになるが、すべてのコンパイラで問題なく使えるとは限らない。プリコンパイルヘッダが gcc でまともに動くようになったのは比較的最近であり、ゲーム機の開発環境が提供するコンパイラがどの程度の完成度かは前もってはわからないのが普通である。^{*1}

そういうわけで、クラスの数だけ h と cpp のペアが存在せねばならない、というような固定観念は捨てた方が幸せになれる。そうした方がいい時もあるが、そうでない方がいい時もある。少なくとも SegaLib においては cpp は減らした方が幸せが大きい。すでに書いた通り、PC 版のフルビルド時間はわずか 15 秒である。そして、削ろうと思えばこれを半分にすることも不可能ではない。

なお、クラス間の相互参照さえなければ、理論上 cpp は 1 つにできるが、実際には相互参照は避けられないし、どの cpp に何を置けばわかりやすいか、ということを考えれば、あまり減らすわけにも行かない。ビルド時間を減らす努力が割に合う範囲で減らせば、それで十分である。実際 10 秒を切れれば、それ以上速くしても意味はないだろうし、そうでなくてもすでに 15 秒になっているものをさらに短くしようとは思わない。逆に 3 分を超えているならやる価値はある。3 分間の待ち時間が一日に 10 回以上あるなら、それは生産性を大きく損なっていると言えるだろう。

4.1.4 pImpl イディオム

使用者から見えるクラスの h ファイルは機種非依存にしたい。例えば Texture クラスの中身は機種によって全く異なるが、それは使用者にとっては関係のないことである。関係のないことを使用者に見える h ファイルに書くのは使用者にとっては邪魔なだけである。そこで、Texture の中身が GLuint であろうが、IDirect3DTexture9*であろうと、それが h ファイルの中に書かれないようにする方法が必要になる。

よく知られるやり方は pImpl イディオムと呼ばれ、以下のようなものである。

```
class T{
    ... (いろいろ) ...
private:
    class Impl;
    Impl* pImpl;
};
```

Impl というクラス内クラスのポインタだけを持つ。T::Impl の定義は実装の中に入れる。例

^{*1} このことは複雑な言語機能を使う気を失わせる。実際、私は例外や RTTI を PC 以外では使わない。

例えば Texture であれば、Texture::Impl を他のクラスから参照しないのであれば、Texture.cpp に書けばいい。cpp にクラス定義を書いて悪い理由はない。SegaLib の場合は Texture::Impl が他のクラスから参照されているので、core/src 以下にある TextureImpl.h に書かれている。このヘッダは使用者には配布されない。そしてその Impl クラスの中に GLuint や IDirect3DTexture9 を入れればいい。Texture.h は機種共通だが、TextureImpl.h は機種の数だけある。

この書き方をする事で、Texture.h にはシステムヘッダをインクルードする必要がなくなる。そうでなくては h ファイルを機種非依存にできない。そして、その結果アプリケーション側のビルドが高速化する。PowerSmash3 はライブラリの h ファイルからシステムの h までインクルードしていたため、アプリケーションのビルド時間が大変なことになっていた。並列ビルドなしでは 30 分を超えることもあったくらいである。しかし、PowerSmash4 では特に遅い機種であっても 5 分は超えない。測定してみたところ 4 分 23 秒だった。しかも、これはビルド高速化の工夫を何もしていない状態である。512 個も cpp ファイルがあり、クラスを作りすぎていたり、ヘッダ実装を使える場所でも cpp に実装していたりする。適切に工夫を凝らせば 2 分以内は容易に達成できるだろう。

なお、この方式にはインライン化できなくなるという欠点がある。使用頻度によっては、仮にシステムヘッダが露出したとしてもインライン化可能な作りさせねばならないこともあり得る。しかし現状そこまでせねばならないのは行列クラスくらいである。4 次元行列の上 3 行を格納する Matrix34 クラスは極めて使用頻度が高く、その乗算は機種別に SIMD 最適化が施されている。そのため SIMD 用組み込み関数を使えるようにするためのシステムヘッダのインクルードが必要であり、これをインライン関数にすれば、システムヘッダが使用者側に露出する。しかし、この程度であれば害は少ない。グラフィックスやサウンドのように巨大なシステムのヘッダさえ隠蔽できれば、アプリケーションのビルド時間が大幅に伸びることはない。

ちなみに、OpenGL を使う機種では、glex.h をインクルードする cpp のコンパイルは顕著に遅くなる。1 ファイルで 300KB あるいは 400KB にも及ぶからである。PowerSmash3 のビルド時間が長かったことの一因はそれで、たどってみると全ての cpp が glex.h につながっていた。もっとも、一部のライブラリがマクロとテンプレートの化物として実装されていたのが最大の原因であり、glex.h だけが悪かったわけではない。また、STL の使用もコンパイル時間を大幅に伸ばす。

4.1.5 コア外ライブラリ

ここまで SegaLib と言ってきた部分は必須だが、別にアプリケーションが取捨選択できるモジュールがいくつかある。これらはライブラリファイルが別であり、移植の際には別途手間をかけねばならない。「コアに入れてない機能は勝手に外部ライブラリをつくれ」というのが

SegaLib の基本方針だが、それにあまり手間がかかっては皆に恨まれる結果になる。そこで、実験として、丁度必要になって作った交差判定モジュールは `cpp` をひとつも作らないで実装を行った。全てが `h` に実装されているので、インクルードするだけで使え、何もリンクしなくていい。`vcproj` がそもそも存在しない。新機種への移植の手間はバグさえなければゼロである。

また、グローバル変数やクラススタティック変数が存在するなどして `cpp` が必要になるケースでも、ライブラリファイルを作らない作りは可能である。可能な限り `h` で実装して、どうしてもない部分だけ `cpp` に入れ、その `cpp` をアプリケーションに放り込んで一緒にビルドすれば良い。このやり方であればライブラリの移植コストや保守コストが大幅に下げられる。社内ライブラリは実装を隠す必要がないのだから、何から何まで売り物のライブラリを真似しなくてもいいのである。社内ライブラリには「ソースが丸見えでも良い」という極めて強力な利点があることを思い起こしていただきたい。

4.2 GPU の抽象化

グラフィックスは抽象化すべき最大の機能であり、ライブラリを作る意味の大半がここにあると言っても過言ではない。しかし、一見似たような絵が出ていても機械によってその構造はまちまちであり、得意なやり方も異なる。一つのアプリケーションコードでありながら全ての機種で最適な性能が出る、などということはあるにない。最大限に性能を生かしたいなら、絵作りのレベルから機械別に行うべきであり、したがって素材もプログラムも全く異なるものにするべきである。

しかし、それをやる余裕はない。少なくとも `PowerSmash4` にはない。

グラフィックス部分を設計する際の出発点はここである。全ては、抽象化による省力化と、性能の最大利用の間のバランスをどこに取るかという葛藤の中で進めていくことになる。

4.2.1 描画エンジンでないことの不利

SegaLib はすでに述べたように、`DirectX` や `OpenGL` に当たる部分しか持たず、シェーダをセットする関数や、テクスチャをセットする関数などがバラバラに存在する。使用者にとっての自由度は高いが、シェーダを書く仕事は使用者がやらねばならない。シェーダを書けるプログラマがアプリケーションチームにいなければ、SegaLib を使うことはその時点で不可能になる。そういったチームに提供するために、上に描画エンジンをかぶせる計画もなくはないが、それはそういったチームが現れてからのことになる。

もう一つ問題なのは、ライブラリ内で最適化する自由度が低いことである。

例えば、GPU の構造によっては描画に使うシェーダがわからないとシェーダ定数をセットできない場合がある。シェーダ定数をメモリ内のバッファに書きこむ形式で、シェーダがどの

定数をどこに配置するかの情報を持っている場合がこれに当たる。素直に作れば、`setShader()` を呼んでからでないで `setShaderConstant()` できない、ということになるであろう。しかし、機種によっては呼び出し順に制約がある、というのでは「どの機種でも同じように動く」とは言えまいし、学びやすさも損なわれる。まして、関数を呼ぶことはできるが、順序が違えば機種によって結果が異なる、というような動作は論外である。このような動作は相当に直しにくいバグの原因となる。そこで、即座にプラットフォームの SDK 関数を呼ばず値を覚えておいて必要な情報が全て揃ってからまとめて SDK 関数を呼ぶような設計にせざるを得ない。結果、コードは肥大化し、性能も落ちる。

もし SegaLib のグラフィクス部分をもっと上位層まで含んでいれば、つまりシェーダなどの描画設計を内部で持ち、`setTexture()` や `setShader()` などの関数がなく、`drawModel()` と書けば絵が出るようなライブラリであったならば、GPU の都合は容易く隠蔽でき、機種ごとに最適な実装をすることができる。しかし、そうしてしまえばシェーダを私が書かねばならず、また、アプリケーション側で独自に絵を調整したり、性能と画質のバランスを取ったりすることが困難になる。PowerSmash4 の絵作りを設計するのは私ではないのでシェーダは書けないし、SegaLib が PowerSmash4 以外のゲームに使われる可能性をつぶしてしまうことにもなりかねない。それゆえ、SegaLib においてはその道は採らなかった。シェーダは SegaLib の外にあり、こればかりは機種専用を書いて、機種ごとにコンパイラツールを使ってコンパイルしてもらおう他ない。ただし、PowerSmash4 ではわずかな `ifdef` で分岐する程度で済んでおり、PS3 でも 360 でも同じシェーダのソースファイルを使っている。

したがって、SegaLib は CPU 負荷の面では最適とは言えない部分がかかなりある。ただし、たいていの場合最終的にできることを規定するのは CPU であるよりは GPU であることの方が多く、多少 CPU 処理に無駄があったとしても大勢に影響はない。

前計算バッチ

ただし、SegaLib ではこの問題を軽減するために、あるバッチ（一回の描画コマンドで扱う範囲）に必要な情報を前もって全て設定しておくクラスを追加している。シェーダはコンストラクタで渡すので、シェーダ定数は必ずシェーダがわかっている状況で設定することになり、順序が問題になることはない。そして、前もってできる計算を全て済ませておく事ができ、高速化する。例えば定数類は SegaLib の `enum` から各プラットフォームの定数に変換せねばならないが、これを初期化時に行ってしまうために描画の度にかかっていたオーバーヘッドがなくなって高速化する。

ただし、現状動的に中身を書き換える頂点バッファやテクスチャを使用できない、シェーダ定数以外のパラメータは初期化後には変更できない、一つの前計算バッチは 1 フレームに 1 回しか使えない、などの制約をかかえている。また、メモリもいささか消費する。性能の劣化なくこれらの問題をを解決できるならば、そもそもこれ以外の手段で描画できないようにしてし

まうかもしれない。そうすればグラフィックスのインターフェイスがかなり小さくなる。

4.2.2 GPU の個性

GPU の個性のうち特に問題になるものが三つある。

一つはシェーダの有無である。シェーダが使えるか使えないかは他のどんな差異も霞んで見えるほど大きな差異であり、Wii や低性能の携帯電話と、それ以外の現代的な機械を同じライブラリで扱うに際して最大の障害になる。

次に、レンダーターゲットがどこに置かれるかが問題になる。描画するレンダーターゲットが大きなメモリの一部にあるのか、レンダーターゲット専用のメモリにあるのかは設計に大きな影響を与える。

最後に、シェーダがある場合に限るが、シェーダ定数の扱いである。これが実は GPU によって、あるいはグラフィックスの API によって全く異なる。これを抽象化する方法はよくよく考えねばならない。

シェーダの有無

シェーダがないハードウェアはすでに絶滅危惧種であるように思えるかもしれないが、実は結構な数存在しており、相手にする市場によってはまだまだ無視できない。

さて、シェーダが現れる前の描画を思い出してみよう。描画にプログラブルな部分が少ないか、あるいは全く無いため、基本的に関数と呼んでパラメータを設定することによって絵を変えるしかない。このため、このような GPU を使う際にはシェーダを使う場合とは比べものにならないほど関数の種類が多くなる。しかも、どのような計算が可能なのかは完全に GPU 依存である。

もし今回のライブラリが相手にするハードウェアが全てシェーダを持たないのであれば、ディフューズライトがどうだの、スペキュラがどうだのといった基本的なモデルを想定してインターフェイスを共通化してしまうこともできなくはない。しかしそれではただでも少ない自由度が減りすぎるし、そもそも今回そのような手を使う必要はなかった。シェーダを持たないハードウェアは一つしかなく、また、今後そのようなハードウェアが増えてくる可能性も低いからである。そういうわけで、完全に専用のインターフェイスを別に用意することにした。強いて言えば OpenGL のバージョン 1 しか動かない電話機も相手にする可能性はあるが、その時はまた別途用意すればいい。

なお、どうせ専用のインターフェイスを作るくらいなら、最初から SDK を直接触ってもらえばいいのではないかと思われるかもしれない。確かにそうすればライブラリ側の作業が減り、保守性も上がり、オーバーヘッドもなくなり、自由度も高まる。しかしながら、表示バッファのフリップなどの各機種共通の機能をその機種だけ提供できなくなるし、ライブラリ側で

持っているデバッグ描画機能や画面クリアの際には GPU の状態を知る必要があり、整合性を取るのが困難になる。例えばアルファブレンドが有効なのか、Z テストは有効なのか、などなどが全てわからないことになり、全部の設定をやりなおさねばならなくなるのである。

そこで、性能的に不利であることは承知で、使う SDK 関数に対してラッピング関数を用意することにし、使用者にはそれを呼んでもらうことにした。例えば、もともとの SDK が提供する関数が `glBindTexture` だったとすれば、GL というラッパクラスを用意して、`GL::bindTexture` を用意し、また、`GL_TEXTURE_2D` のようなマクロ定数があれば、GL クラスの enum として `GL::TEXTURE_2D` を用意する、といった具合である。そして、使用者が新しい SDK 機能を使いたいと言ってくれば、その都度足す。SegaLib の基本方針に従い、前もって全関数を用意することはしていない。使いもしない関数が並んでいるのは邪魔だし、使わないで終わるものをわざわざ作る必要はないからである。アプリケーション側の描画担当は隣に座っており、要求があれば 5 分で提供できる。

むろん、このようなラッピングをした所で、使用者も結局 SDK のマニュアルを読まねばならないことには変わりはない。機種抽象化という目的を考えれば全く無意味なのではあるが、少なくともこうすることで、呼ばれる関数の数を減らし、また、何が呼ばれたのかをライブラリ側で把握することができるため、デバッグを容易にすることはできている。オーバーヘッドが心配されはしたが、結局問題になるほどではなかった。

レンダーターゲットの場所

レンダーターゲットは専用メモリにあるのか、あるいは普通のビデオメモリにあるのか。これによって同じことをやるにしてもやり方は異なる。まず当たり前のことだが、専用メモリにある方が制限は多い。

普通のビデオメモリにレンダーターゲットを置ける場合、描いた絵はいつまでも置いておける。別のレンダーターゲットに切り替えるのも簡単で、単に書き込みアドレスを切り替えるだけでいい。

しかし、専用メモリの場合そうは行かない。レンダーターゲットを切り替えるには、レンダーターゲットメモリからデータをメインメモリに移さねばならない。専用メモリにあるものは次の描画で消えてしまうからだ。大容量のコピーが発生する。例えば 20GB/s の帯域で 1280x720 の 2xMSAA バッファを転送すれば、それだけで 0.3ms かかる。これは決して安いコストではない。したがってレンダーターゲットの切り替えはかなり高くつく。

さらに、途中まで描いてから別のレンダーターゲットに何かを描き、また戻ってくる処理はさらに厄介だ。レンダーターゲットメモリからの移動に加えて、レンダーターゲットメモリへの移動が必要になる。この場合は Z バッファまで必要になるため、なおさら大きい。レンダーターゲットメモリを別に持つ構造の方が制約が厳しい以上、共通化を重視するならこちらに合わせてできることを制限することになる。結果、どこにでも描画できるハードウェアなら当た

り前にできることで、他の機種に合わせてできなくなることになる。

例えば、シャドウマップの処理手順はこのせいで PowerSmash3 と変えざるを得なかった。PowerSmash3 では、4 人のシャドウマップは別々のパスで描画していたが、レンダーターゲットは使いまわすことができた。メインの描画を中断して 1 人目のシャドウマップを描き、またメインに戻ってそのシャドウマップを使い、次に先ほどのシャドウマップ領域に 2 人目のシャドウマップを描き、またメインに戻ってそのシャドウマップを使い、ということを 4 回繰り返していたわけである。しかし、今回はそれをする、専用メモリを持つ機種が割を食う。そこで、4 枚分のメモリを前もって用意してしまい、4 人の影を連続して描いてから、メイン描画にてそれを使うようにした。幸い、大きなバッファを必要とするのは一番大きく映る一人だけで良いということがわかり、これを利用して必要容量を減らすことができたので、容量もあまり増えていない。その意味でこのインターフェイス変更にはなんら問題がなかったわけだが、しかし、設計を変更せねばならなくなったのは確かである。

なおこれは余談だが、専用メモリでなかったとしても、同様にレンダーターゲットの変更が高くつくケースがある。例えば GPU のパイプラインを停止させないと切り替えられないようなケースではコピーするほどではないにせよタダではない。そういうわけで、複数機種に対応する場合には、レンダーターゲットの切り替え回数は極力減らしておくべきである。SegaLib はレンダーターゲットの切り替えの自由度をインターフェイス的にかなり制限しており、アプリケーション側がパス数を増やす気を失うように仕向けている。例えば、プライマリレンダーターゲットに一回でも描画すると、その後のレンダーターゲット変更は許さない。また、フレーム内で同じレンダーターゲットを複数回使うことも禁止している。

シェーダ定数

DirectX9 でいじっていると気づかないが、実はシェーダ定数の扱いは GPU によって全く違う。DirectX9 では「何番レジスタに何を送る」というような指定の仕方ができるが、場合によってはレジスタという概念がないケースもある。例えば OpenGL ではシェーダに対して「○○という名前の定数はどこにありますか」と聞くと、「このハンドルを使ってアクセスしてください」とハンドルを渡される。このハンドルが何者かは全くわからない。もう少し具体的に、何かのメモリのオフセットが返ってくる SDK もある。

いずれにしても、インターフェイスを DirectX9 的にしてしまうと一旦ライブラリ内で仮想レジスタ番号を割り当てて後で再構成して送るような手間が必要になり、効率が悪い。しかも、番号でアクセスできるハードウェアに対して、番号を使わないことは不利である。番号指定であれば、「前のシェーダでは 5 番レジスタに 1 を書き込んだ。だから今回 5 番は書き換えなくていい」というような最適化が可能になる。SegaLib の中でそのような最適化をすることはできるが、SegaLib のインターフェイスが番号指定でなければアプリ側でその工夫を行うことはできない。最適化は上流でやるほど効率がいいが、インターフェイスの共通化によってそ

れができなくなる。

また、シェーダ定数を GPU に DMA されるコマンド列に埋め込むのか、または全く別のメモリに書きこんでおくのかによっても最適な設計は異なる。メモリに書きこんでおけば GPU が直接そこを見に来る、という場合、描画が終了するまでそのメモリを書き換えることはできない。描画終了を検出する手間をかけたくなければ、1 フレーム分の全ての描画におけるシェーダ定数用のメモリを前もって確保しておくべきだ、ということになる。本来こういう場合には使用者に明示的にメモリを確保してもらった方が性能を出しやすいが、インターフェイスを共通化するならばそんなことはできない。しかし、シェーダ定数の転送はかなりの比率を占める重い処理であり、機種別にコードを書くことを許すか許さないかは重大な問題になりうる。

PowerSmash4 を発売するまでの SegaLib においては、一部の機種では特別な関数を用意し、アプリケーション側に `ifdef` が入ることを許容していた。現在は抜本的な設計の見直しを行った結果、全機種同じコードでにそれなりに性能を出すことが可能な設計にはなっている。ただし、例によって SegaLib は「今使う機能以外は保守しない」状況になっており、360、Wii、PS3 の 3 機種は現在保守範囲外である。今はもう動かない。必要になったらまた実装し直すだろう。

4.3 CPU の抽象化

ゲーム機によって CPU の性能や構成はさまざまである。そこそこ速い汎用 CPU が複数あるものもあれば、特殊なコプロセッサが別途入っているものもある。これらの複数の CPU を使いこなしてこそハードウェアの性能を使い切れるわけだ。

しかし、PowerSmash4 の開発、そして SegaLib の設計においては、ここはほぼ切り捨てた。いわゆる「作法に従ってゲームをジョブの集合として実装すれば、CPU の数に応じて勝手に並列処理される」というような作り方は導入しなかった。

なにせまだシングルコアの CPU を持つハードウェアがあり、その性能が一番低い以上、それに合わせるのが最適と考えたからである。確かに、CPU の数に応じて動作が変わるような設計にしておけば、1 コアでも動くだろう。しかし、そのようなシステムは複数コアでこそ威力を発揮し、1 コアしかないマシンでは無駄になる。ただでも性能が不足するであろうマシンに割を食わせるのは危険である。

また、アプリケーションプログラムにマルチスレッドの経験が足りているかはその時点では未知数だった。マルチスレッドはちょっと概念を学んだ程度で使いこなすことはできず、一見動いていても後から問題が発生する可能性がある諸刃の剣である。アプリケーションの作り方の根本にこれを組み込むような設計は危険と判断した。

さらに、一部の機種では単にスレッドに投げれば並列化できるというものでもなく、コード

の書き方から変えなくてはならなかった。これをアプリケーション側に強いるのは妥当とは言えない。

以上から、SegaLib 上のアプリケーションは伝統的なシングルスレッドアプリケーションとして書かれることを想定することとした。そして、とりわけ性能を必要とする処理のうち定型化できるものは SegaLib 側で機種別の実装を提供した。例えばスキニングがそれである。残る非定型的な処理、つまりアプリケーション固有の処理でライブラリ側ではどうにもならないものに関しては、SegaLib が用意したスレッドプールに投げて並列化してもらったこととした。

すなわち、SegaLib アプリケーションは特段の事情がない限り、自前でスレッドを立てない、という前提である。むろんスレッドを立てざるを得ない処理はあるが、それらは高速化を目的とはしない。非同期化あるいはサービスの常駐を目的とする。

4.3.1 スレッドプール

スレッドプールとは、前もって立てておいたスレッドに関数ポインタを渡して実行してもらう仕組みである。アプリケーションコードの並列化はこのスレッドプールを使うのが基本である。

例えば、4 人のテニス選手はそれぞれアニメーションの計算を行うが、場合によってはこれが大きな負荷になる。そこで、アプリケーションはこれらをスレッドプール用のジョブの体裁に整え、SegaLib に渡す。

```
class MyJob : public Sega::ThreadPool::Job{
public:
    void operator()(){ ... 処理... }
};

MyJob job( ... ); //ジョブ生成
Sega::ThreadPool::add( &job ); //処理開始
job.wait(); //処理待ち
```

スレッドプールにいくつスレッドがあるかは、SegaLib 内部で機種別に定義されており、アプリケーション側が知る術はない。そもそも本当にスレッドで実行されるかどうかすら定かではない。実際、シングルコアマシンでは投げた瞬間にその場で実行され、スレッドを用いない。こうすることで、特に問題になる処理は並列化の恩恵にあずかれ、シングルコアであってもさほどの無駄なく処理できる。

スレッドプールは CPU の数が不定な状況で並列化の恩恵を受けるための一番楽な方法である。今後、1 コアあたりの処理速度が向上せずひたすらコアが増えていく方向に行くのだとすれば、こうした作り方が有利になるだろう。ただし、同期オーバーヘッドの問題もあるし、本

当にそれほどコアが増えるのかについては現状なんとも言えない。

なお、スレッドプールの実装は極めてシンプルである。中にはキューとセマフォが入っており、`add()` でキューに追加し、セマフォをインクリメントする。セマフォのインクリメントによってワーカースレッドが起きだして、キューからジョブを取得して実行し、終了フラグを立てる。これ以外のことはしていない。優先度や依存関係などは一切ない。必要になってから考えることにする。100 人の兵士の AI とアニメーションと描画が全部それぞれジョブになって並列実行される、というくらいに活用される状況にならない限り、そのような高度な機構は必要ない。

将来の多コア時代にどう対応するか

多コア時代は本当に来るのであろうか？

確かに理論的には来るはずである。1GHz のコア 1 つよりも、500MHz のコア 2 つの方が電気を食わず、現在の CPU はトランジスタ数ではなく消費電力によって規定されている。理論的には、性能が低いコアが大量にある状態になるのがもっとも効率が良い。しかし、そもそも並列化したいような用途には GPU を汎用化する方が効率が良く、CPU と GPU の間にもう一種類中途半端なプロセッサを設けるアプローチはプログラマにかかる負担が大きいという問題がある。

さて、SegaLib は現在のライブラリであり、未来のライブラリではない。SegaLib にとっての現在とは、およそ今後 3 年までの範囲である。今後 3 年間を考えた場合、コア数の下限は 1 から 2 にはなるだろうが、4 になるかはかなり怪しい。上限は「普通に普及するマシン」に限定すればせいぜいが 8 である。

この程度の数であれば、メインスレッド + スレッドプール、という単純なアプローチでもおそらくは十分であろう。コア数が増えるに従って、ゲームの作りを徐々にスレッドプール向けに直していってもらうことになる。その過程で、ジョブの依存関係を処理できるように拡張する必要が出てくるかもしれない。しかし、今のところは不要だし、有害にもなりうる。ジョブ同士の依存関係にバグがあれば停止してしまいかねないし、下手な依存関係の設定は性能を著しく損なうことにもなる。可能な限り投げて終了を待つだけの作りにする方が良い。

4.4 アニメーション

SegaLib のアニメーション機能は、SegaLib の基本理念から言えば余計な存在である。機種非依存であり、アプリケーション側で書くこともできるし、あるいは SegaLib とは別のライブラリとすることもできる。しかし、前述のように、SegaLib の一部として用意することで使用を半ば義務づけた。これは「アプリケーションの作り方を規定するライブラリ」であることの一例である。

4.4.1 どのような機能が

実のところ、SegaLib のアニメーション機能は、アニメーション機能と呼ぶのもおこがましいほど低レベルである。SegaLib が持っているのは、とある XML 形式で書かれた「スカラ値あるいはクォータニオン値の時間変化データ」に近似と量子化による圧縮をかけてバイナリファイルを吐き出す部分と、それを展開して求めに応じて元の値を復元する部分だけだ。

バイナリ化するデータは、

```
<AnimationContainer>
  <Tree>
    <TreeNode>
      ...
    </TreeNode>
  </Tree>
  ...
  <Animation>
    <CurveSet>
      <Curve>
        <Key time value/>
        ...
      </Curve>
      ...
    </CurveSet>
    ...
  </Animation>
  ...
</AnimationContainer>
```

という単純な構造である。「時刻 0 では値が 5 だった」という単純なデータである Key を複数持った Curve、それをまとめた CurveSet、それをまとめた Animation、そして別途木構造を規定する Tree、そして全体を含む AnimationContainer である。

ランタイム側が持つ機能は、つまるところ「何番目の Animation の、何番目の CurveSet の、何番目の Curve の時刻いくつの値はいくつ？」と聞くことだけである。これをどう組合わせて実際のアニメーションを実現するかは、アプリケーション側に委ねられている。play() と呼ぶと毎フレームアニメーションが更新されて動く、というようなものを作るには、かなりのコードを必要とするわけだ。

何故このような実装にしたかと言えば、SegaLib 側がアプリケーションの作り方に関して置く前提はさして多くないからである。アニメーションの制作工程も、アニメーションの計算方法も、アプリケーションによって設計は異なる。とりわけ IK が絡むと共通化は困難だ。標準的な用法を用意してそれ以外はクラス継承か何かでカスタマイズしてもらおう、という設計も

あるだろうが、それをすれば標準は一切使われず、いきなりカスタマイズされるだけに終わるだろう。

実際 PowerSmash3 において使ったライブラリには標準的な用法が用意されていたが、全く使わなかったのである。使われない標準機能は無駄なばかりか有害である。そして、そもそも標準機能なるものを作る時間は私にはなかった。

そして何よりも重要なことに、私はアニメーションは正直素人である。しかし、データのバイナリ化や圧縮に関しては比較的専門だ。正直自分がこの道で一流のスキルを持っているなどとは思えないが、さしあたって私がやるのが一番コストが安く、出来が良くなる可能性が高かった。下手にアプリケーション側で苦手な人が無理やり書くようなことになるくらいなら、標準として提供してしまおうと思ったわけである。その意味で、いずれアニメーションは SegaLib の外部に出す可能性もある。

ただし、アニメーションの内部にはいくらか機種依存実装がある。SIMD 最適化や、機種によっては問題になるロードヒットストア対策、また別スレッドや別種プロセッサを使った非同期並列計算などを提供しており、これを鑑みると SegaLib として抱え込む以外の選択肢はなかったかもしれない。SegaLib の考え方としては、外部ライブラリもまた SegaLib アプリケーションであり、そこには機種依存コードがあるべきではないからである。

4.4.2 データ圧縮

SegaLib のアニメーション機能においては、ほとんどの労力をアニメーションデータの圧縮効率を上げることに費やした。というのも、PowerSmash は極めてアニメーションデータの量が多いゲームであり、PowerSmash3 では同時に使用するデータ量が数十メガバイトにも及んでいた。3 はそれでもメモリに入ったので問題なかったが、今回はメモリが少ないハードウェアが混ざっている。もちろん性能が高い機種と低い機種で別のデータを用意する手もあるが、アニメーションに関してはゲームの根幹を成すものでもあり、できれば同じデータで行けるようにしたい。そこで、この部分に関して改良できないか検討してみることにしたわけである。

そして調べてみたところ、PowerSmash3 で使っていたアニメーションライブラリにおいて、データ圧縮はさほど考えられてはいなかったことがわかった。一からきちんと設計すればかなりの削減が見込めるであろうという見込みりが立ったわけである。圧縮の戦略は以下のようになる。

まず、最小のデータ型を使うことだ。曲線データは、誤差が一定範囲に収まるようにデータが間引かれ、間は補間によって表現される。一つのデータは、その点の時刻と関数値、それによって場合によっては微分係数からなる。

```
KeyFrame{
    Integer mTime;
```

```
Real mValue;  
Real mDerivative;  
};
```

従来はこの時刻 (mTime) に全て 2 バイト (unsigned short) を使っていたが、今回は 255 フレーム以下のアニメーションであれば 1 バイトとした。PowerSmash において、ボールを打ったり走ったりするアニメーションはおおむね 4 秒以下である。したがってほとんどのアニメーションでは時刻が 1 バイトで事足りる。これは他のゲームであっても同様であろう。関数値と微分係数においては、1 バイト、2 バイト、4 バイトの 3 種類を試し、所定の最大誤差に収まる範囲でもっとも小さいものを採用した。さらに、後述するようにクォータニオンは 4 スカラでなく 3 スカラで表し、同様に 1,2,4 バイトの 3 種類を試した。圧縮にかかる時間は伸びたが、これは所詮ツールの実行時間であり、複数プロセス同時に走らせるなどの工夫で速度はどうにでもなる。

次に、補間の種類を二種類とした。線形補間と、三次多項式によるスプライン補間である。スプライン補間の方が理論的にはデータの数を減らせるが、微分値を持たねばならないためにデータの容量が減るとは限らない。両方試してデータが小さくなる方を選ぶこととした。

なお、補間しないモードは持たない。よって、「時刻 100 未満は 0、時刻 100 から 1」というデータは、厳密には表現できず、「時刻 0-99 まで 0、時刻 99-100 は 0 から 1 で補間、時刻 100 から 1」となってしまう。しかし、これはアプリケーション側が時刻を整数で扱うなり、出てきた値を整数に切り捨てるなりすればいいだけである。SegaLib の制約であるとして納得してもらうこととした。もともと圧縮関数への入力時刻 0,1,2,3... における値のリストであり、整数時刻におけるデータしか含まない。その間の値がどのようなものであるかはわからないのである。

量子化

データを 1 バイトまたは 2 バイトで表す場合、浮動小数点数ではなく固定小数点数となる。0 ならば 0、255 であれば 1、のように値の範囲を決めてしまえば楽だが、実際にはどのような範囲のデータが来るかはわからない。そこで、最小値と最大値をまず探し、それが 0 から 255、あるいは 0 から 65535 になるように線形変換をかけて量子化している。したがって、デコードのためには線形変換の逆変換を行うためのパラメータが必要であり、これをカーブごとに保持している。

クォータニオンの圧縮

クォータニオンは 4 つのスカラを持つ。しかし、4 つのスカラを持つことは容量的には最小ではない。回転用のクォータニオンは 4 要素の二乗の和が 1 であり、第 4 要素については符号

だけがわかれば復元できる。そのため、理論的には3要素と1ビットあれば良い。実際には3要素のどれかから最下位ビットをもらってきてそこに符号を入れることになる。確かに理論上はこれでうまく行くはずである。実際線形補間までを実装したが、それなりに使えた。

しかし、問題はスプライン補間である。クォータニオンのスプライン補間は Game Programming Gems の1巻に書かれていることもあってそれなりに有名なようではあるが、数学的な意味を理解できる人はそうそうおるまいというくらいややこしい。しかも、計算は結構な負荷になる。考えてみれば今求めているのは数学的な厳密さではなく、データを減らせるかどうかだ。補間の性質すらどうでも良い。線形補間においても、球面線形補間である必要は必ずしもなく、データが小さくなるのであればただの4要素独立の線形補間でもかまわないのである。最短距離を通るかどうかなどは全くもってどうでも良い。であるならば、もっと単純な方法でも一向にかまわないわけだ。つまり、4要素それぞれに独立にスカラ用のスプライン補間をかけても一向にかまわない。3要素と符号であれば、3要素だけ補間し、補間結果と符号から第四要素を計算すれば良い。実際それで正しく動作する。ただし、残念ながら圧縮率はさほど良くなかった。やはり最短距離を通る綺麗な補間である方が圧縮率が高い傾向があるということだろう。

最終的にはクォータニオンは全てその対数形式で格納することにした。クォータニオンの対数形式とはある単位クォータニオン q に対して、 $\ln(q)$ のことであり、それを v とするならば、 $v = \ln(q)$ すなわち、 $\exp(v) = q$ となるような v である。私のように数学に疎い人ならば、何のこともわからないまい。私もそこに納得するのに二週間以上を要した。しかしなんとなく納得するだけならば、複素数と比べれば良い。例えば、単位複素数 c に対して、 $\ln(c)$ というものを考え、それを a とするならば、 $a = \ln(c)$ すなわち、 $\exp(a) = c$ となるような a である。オイラーの公式から、 $\exp(i\theta) = c$ と書けるはずであり、 $a = i\theta$ だ。つまり、 a とは虚数単位に回転角を掛けたものである。これを指して「複素数の対数化したものだ」と言うことはできるだろう。単位複素数を対数化すると、実数部は0になって、1つのスカラで格納できるようになる。

クォータニオンに関しても同じだとすれば、 v は回転角と3次元の虚数単位を掛けたものだろう。 $i\theta$ を3次元に持ってきたようなものであり、実際3つのスカラで表せる。複素数の時と同様に、単位クォータニオンを対数化すると実数部が0になって、3つのスカラで表せるのである。

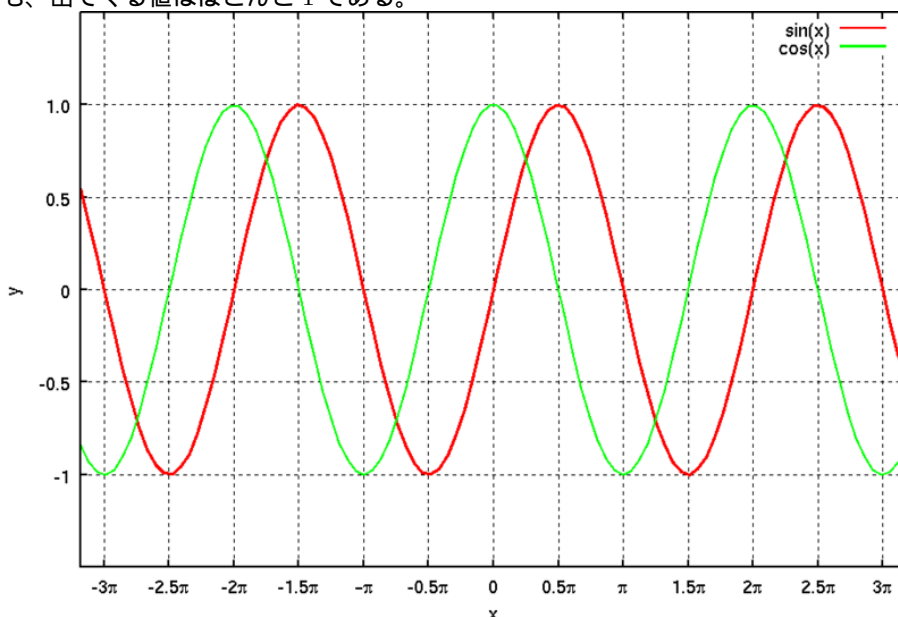
考えてみれば、オイラー角で姿勢を表せば3スカラで足りるわけで、本来姿勢を表現するだけなら3スカラで足りるはずなのである。オイラー角ではクォータニオンへの変換に手間がかかりすぎて採用できないが、この対数クォータニオンにはその欠点がない。

そういうわけで、クォータニオンデータの容量は4スカラ格納するのに比べて25%削減された。また、対数化クォータニオンには、単純に線形補間しても球面線形補間にそこそ近い感じに補間が行われるという利点がある。角度みたいなものであることを考えれば、これは納

得しやすい。時刻0で0度、時刻100で100度なら、時刻50では50度だろう。3次元のわけのわからない数学においても、だいたいそれに近い感じの結果が得られる。理由は知らない。結果、圧縮率を上げることができた。とは言え、どれくらい減ったかのデータを厳密に取ったわけではない。そもそもデータに依存するので、この方法が常に最も良いと言うつもりはない。こういう選択肢があり、たまたま現在の SegaLib ではこれを採用している、というだけのことである。将来暇になればもう一度全方式を比較検討することもあるだろう。

現在のところ、線形補間是对数状態を解いてから球面線形補間、スプライン補間是对数化したままで3要素独立にスプライン補間してからクォータニオンに戻す、という計算を経ている。かなりの計算量だが、クォータニオンの4要素に対して独立にカーブを持ってバラバラに計算するよりはまだ速く、容量も減る。アニメーションデータの半分以上は回転データであることを考えれば、クォータニオンに対する圧縮率と、計算速度は極めて重要になるのである。なお、PowerSmash3の時には回転を特別扱わず、単に4つのカーブとして扱っていた。

ただし、この方法には問題もある。誤差だ。対数化はツール上でのことなので double 精度でやっているが、結局 float 以下の精度に落とされて格納される。ここで大きな誤差が生じる。というのも、対数化の際に通す $\cos()$ によって、本来大きく異なる値が似たような値に変換されてしまうからだ。0度付近の \cos のグラフを思い浮かべていただきたい。角度が多少変わっても、出てくる値はほとんど1である。



したがって、4要素のクォータニオンをそのまま格納するのに比べればかなり精度が落ちる。もし関節がかなりの数連なっている場合、回転の小さな誤差も末端の関節ではかなり増幅されてしまう。これによって、例えば指先が不規則に揺れるようなことが起こる可能性はある。現

在 PowerSmash4 程度の関節数であれば問題は起きていないが、将来たとえば蛇の化物が出てくるようなゲームを作れば問題になる可能性がある。誤差要求を満たさない場合には対数化せずに格納したり、あるいは対数化やその後の量子化を改良することで改良できればと思うが、今のところは全く何もしていない。誤差が生じても無視している。

なお、クォータニオンの球面線形補間時のために $\sin(x)/x$ を計算する関数を別に作っておくと良い。sin のテイラー展開は奇数次しか含まないため、 x で綺麗に割ることができる。標準の $\sin()$ を呼んでから x で割る場合、 x が 0 の場合を特別扱いせねばならず、速度が落ちる上に面倒であるが、この問題がなくなる。これに関しては数学機能に関する項にて詳しく述べる。

曲線近似

曲線のスプライン近似は比較的単純なアルゴリズムで行っている。まず始点 A と、そこから N 点離れた点 B と、その次の点 C を取る。N は最初 1 とする。この 3 点を通る 3 次多項式を計算し、その後、範囲内において、元のデータと多項式の計算値の誤差を求める。この誤差が許容範囲に入っているならば N を +1 して繰り返す。誤差が許容範囲を超えたら、超える直前の多項式から、B における微分値を求めて B の関数値と共に格納する。始点を B とする。以上を最後まで繰り返す。

もっと良い方法も考えられるし、理論上最適な方法も存在するが、ツールとは言え速度は重要であるし、実装にかけられる時間制約の問題や、保守性を考えれば、あまり複雑な手法は取りたくない。幸い、この方法でも PowerSmash3 で使っていた方式よりもはるかに圧縮率が高まったため、これで良しとした。

データ配列の格納

データは時刻、関数値、微分係数からなる。しかしながら、これらを構造体にまとめてその配列としたのでは、時刻とそれ以外のデータ型が独立に変わるこの実装においては不便である。

```
struct Key{ Integer time, Real value, Real derivative; }
Key* mKeys;
```

Integer が 2 種類、Real が 6 種類 (スカラ 3、クォータニオン 3) あるので 12 種類の Key 型ができるわけだ。そこで、時刻は時刻のみで配列をなし、関数値と微分値は別の配列に交互に格納するようにした。

```
void* mTimes;
void* mValuesAndDerivatives;
```

アラインメントの問題もなく、それぞれの型が異なる場合にも問題が起きない。

さらに、こうすることで理論上は性能も改善される。時刻 t における値を計算するには、復元に用いる区間を検索せねばならない。時刻 0,5,10 とデータがあって、時刻 4 のデータが欲しければ、時刻 0 と 5 を持つデータで補間すれば良い。この検索には時刻だけがあれば良く、時刻だけが別配列になっていることで読み込むメモリの量が減ってキャッシュミスしにくくなり性能が向上する。ただし、理論上のことであり、確認はしていない。

なお、この検索には二分検索を用いている。計算要求の度に +1 ずつ時刻が増えていく場合がほとんどなので、前の検索結果を覚えておいてその位置から線形検索した方が速いこともあり得るのだが、それには前の検索結果を覚えておかなければならずマルチスレッドの場合に問題が起こるし、そもそもリードオンリーで計算できる方が実装が簡単になる。そして、覚えておくのにメモリも余分に必要になる。メモリ容量を速度よりも重視するのが SegaLib の基本方針であり、このような方法は取りたくない。そういうわけで、毎回二分検索する実装となっている。

実行時データ構造の省メモリ化

データは 1 バイトだったり 2 バイトだったりし、補間方法も線形補間だったりスプラインだったりする。こういう場合に綺麗に書く方法として普通に考えるのは、Curve という基底クラスを派生したそれぞれのデータ型専用 Curve 型を作ることであろう。時刻が 1 バイトで値が 2 バイトなら CurveT1V2、時刻が 2 バイトで値が float 要素のクォータニオンなら CurveT2Q4、というような具合である。

しかし、これをやると計算関数が仮想関数になりただでも遅いアニメーション計算を余計に遅くすることになる。さらに Curve の配列を作る時に、ポインタ配列として定義して、Curve はそれぞれ new することになる。これではメモリのオーバーヘッドも大きい。

```
Curve** mCurves;
mCurves = new Curve*[ curveCount ];
for ( ... ){
    mCurves[ i ] = createCurve(); //ファクトリ関数。中に new がある。
}
```

そういう事情により、Curve クラスは一つしかなく、内部のメンバ変数によって分岐させつつ、可能な範囲で union を使うことで容量を削減している。

```
Curve* mCurves;
mCurves = new Curve[ curveCount ];
```

また、データが一つしかない場合には時刻や値の配列を作らず、通常それらのポインタが入る場所に関数値や微分値を格納している。

```

union{
    void* mTimes;
    float mValueWhenOneKey;
};
union{
    void* mValuesAndDerivatives;
    float mDerivativeWhenOneKey;
};

```

データが一つしかない場合は Curve そのものを作らないという選択肢もあったが、構造が複雑になるのでやめておいた。

4.4.3 計算

計算関数は、データ型や補間タイプによって分岐して、必要な計算を行うだけである。コードは複雑に見えるが、単に場合分けが多く、またデータのデコードが面倒なだけで、見かけほど複雑ではない。

ところで、この計算をする際に非常に問題になることがある。

ロードヒットストア問題

CPU によっては、メモリに書いた直後にそれを読み出すと数十サイクルの間 CPU が止まってしまうという性質を持つ。これをロードヒットストア (load-hit-store) と言う。load が store に当たるわけだ。これが一番起こるのは、int と float のキャストにおいてである。

ある種の CPU では、int 用のレジスタと float 用のレジスタの連絡経路がなく、一旦メモリに書きこまない限り受け渡しができない。そのために、int を float に変換するにも、float を int に変換するにも、このロードヒットストアが発生する。これが何故問題になるかと言えば、データの検索の際にどうしてもこのキャストが必要になるからだ。

計算関数は、任意の時刻 t における関数値を計算する。 t は float である。何故なら、ゲームがスローになった場合や、アニメーションの速度を微妙に調整した場合などには時刻に整数以外が入ることがあるからである。しかし、アニメーションデータ中の時刻は 1 バイトか 2 バイトの整数である。例えば、時刻 2 と 3 におけるデータがあって、時刻 2.5 の計算を求められれば、2 と 3 の関数値の中点を返すことになる。この時、時刻 2.5 が 2 よりも大きく 3 よりも小さいことを調べるには、格納されている 2 や 3 を float にして 2.5 と比べるか、2.5 を整数化して 2 を作ってから整数型である 2 や 3 と比べるかしかない。前者よりも後者の方がキャストの回数は少ないため、当然のように後者を選ばれる。ここでキャストが一回発生する。さらに、補間の際には、2 と 3 の丁度中間である、ということを知るために、2.5 から 2 を引いた 0.5 を必要とする。2.5 から 2 を引けば小数点以下が出るわけだが、この過程で 2.5 を切り

捨てて整数化して2を作り、これをそのまま float にしてそれを2.5から引くということになる。整数の2を作るのは検索の時にもうやったが、浮動小数点数の2を作るキャストは新しくやらねばならない。

```
int intT = static_cast< int >( time );
//intT を使ってキー検索
float interpolationFactor = static_cast< float >( intT );
//interpolationFactor を使って補間
```

というわけで、補間まで含めると2回の int-float 変換が必要である。正直に書くと2回ともロードヒットストアが発生し、フレームあたり何千回と呼び出す関数であることを考えると馬鹿にならない。

これを軽減するには、整数変換を前もってやっておくことが有効だ。例えば時刻2.5で計算するならば、他のカーブについても2.5で計算することが多かろう。であれば、整数化した2を作るのは1回で良く、ロードヒットストアのダメージを半分にできる。この半分をゼロにしたければ、検索だけ複数のカーブに対して済ませてしまって、その時点で小数部を計算して覚えておき、あとでまとめて補間を行う手がある。書き込んでからある程度時間が経っていれば引っかかることはない。

ただし SegaLib では基本的にはここまではやっていない。余計な計算でかえって遅くなることを危惧したのもある。また、機種によってはロードヒットストアがない別の演算器を使えたり、そもそもCPUがネックになっていなかったりする。であれば、放っておいてもよかろう。

なお、ロードヒットストアが問題になる他の例としては、関数に引数を渡す時に、その引数がメモリに格納された場合がある。呼び出し前にメモリに引数を入れ、呼び出された後メモリからそれを取り出す。これで立派なロードヒットストアである。というわけで、引数はできるだけレジスタのまま渡すことが望ましい。引数の数を増やさないこと、とりわけ構造体やクラスの値渡しを避けることが望ましい。ただし、私は普段から引数が多い関数は書かないし、ユーザ型を値渡しすることもないため、この原因によるロードヒットストアが起きることを確認したことはない。理屈上の話である。

4.5 数学

数学関数、例えば $\sin()$ や $\cos()$ の類は、どんな機械であれ、SDK側が持っている。そもそも、今時C標準ライブラリが入っていないことなんてまずありえない(ただし全機能があるとは思わない方がいい)。加えて、プラットフォームによっては若干の誤差を許容することでさらに高速なバージョンが提供されていることもあるし、4並列で計算するバージョンがあったり

もする。したがって、一機種だけでゲームを作るのであれば、これらを使えば済む話である。

しかし、マルチプラットフォームとなると若干話が変わってくる。

アプリケーションのコードは一つであるべきである。したがって、全機種にある $\sin()$, $\cos()$ を使うのが一番楽だ。この場合、機種によっては存在する高速版を使えない。また、標準の $\sin()$, $\cos()$ がかなり遅いこともある。標準の場合は NaN や Inf を扱えねばならず、精度の要求も厳しいし、手を抜いた実装だと中身が double で float にキャストして返しているだけだったりする。

そこで、SegaLib では試しに自作してみることにした。運良く出来たものは案外高速で、誤差も実用的な範囲に収まった。SegaLib としては最後の 1bit まで正しいことは保証しないが、どうしても精度がほしい場合には標準関数を使えばいいので問題はない。

4.5.1 多項式近似

SegaLib では、使用頻度が高く速度が問題になりがちな \sin と \cos について、自力実装を用意している。さらに、回転行列を作る際のように \sin と \cos が同時に必要な場合も多いので、同時に求める関数を用意している。バラバラにやるよりも速い。

一方 \tan や atan は標準に投げるだけである。これらを 1 フレームに数百回以上呼び出すような処理が現状ないからだ。需要が出てくれば改めて考えるだろう。ただし、 \sin や \cos と違って、下手な実装では標準に劣る速度しか出ない。

また、入力値が特定の範囲に収まっている場合に特化したバージョンをいくつか用意している。これは acos や \cos 、そして $\sin(x)/x$ について用意している。これらはクォータニオン計算に主に使われ、速度の要求が厳しい。そうでなくても、 $-\pi$ から $+\pi$ までしかないとわかっていれば周期の還元を行う手間を省くことができる。ゲームにおいてもかなりのケースではそういう制約を課せるはずである。

さて、 \sin や \cos 、そして asin はテイラー展開で計算できる。ただし、テイラー展開は展開の中心から離れると急速に誤差が増すため、例えば 0 から $\pi/4$ の範囲で、0 を中心にしたテイラー展開で計算すると、0 付近では正確すぎる一方、 $\pi/4$ での誤差が大きくなりすぎる。そこで、チェビシェフ (chebyshev) 多項式を用いて誤差を範囲内で散らすことで、より低い次数の多項式でも高い精度が得られるようにした。例えば SegaLib で使っている近似式は、

$$\begin{aligned}\sin x &= 0.9999999692202622x \\ &\quad - 0.16666650686603645x^3 \\ &\quad + 0.0083320363304474289x^5 \\ &\quad - 0.0001950396312637314x^7 \\ \cos x &= 0.99999997237594876\end{aligned}$$

$$\begin{aligned}
& - 0.49999856557568093x^2 \\
& + 0.04165502090541779x^4 \\
& - 0.0013585843887420873x^6 \\
\arcsin x = & 1.0000000769029826x \\
& + 0.16665163720742029x^3 \\
& + 0.075464795806892628x^5 \\
& + 0.039721134840256107x^7 \\
& + 0.050505363021511584x^9
\end{aligned}$$

といった具合になる。テイラー展開からわずかにずれており、このずれによって 0 付近での誤差を許容する代わりに 0 から離れた点での誤差を軽減させている。このあたりについては、浜田穂積著「近似式のプログラミング」に詳しいので、必要なら参考にすると良いと思うが、我々にとって必要なのは結果として得られる近似式だけであり、この分野を真面目に勉強するのはやりすぎであるように感じる。しかし本気でその本に書いてあることを適用すれば、もう少しだけ改良できる。一度はやってみたのだが効果は微々たるものであり、その係数を計算するツールの保守のことも鑑みて捨てた。

なお、 $\sin(x)/x$ については、

$$\begin{aligned}
\sin x = & 0.99999999692202622 \\
& - 0.16666650686603645x^2 \\
& + 0.0083320363304474289x^4 \\
& - 0.0001950396312637314x^6
\end{aligned}$$

と、 \sin の展開式を 1 次落とすだけで計算できる。乗算が減るため \sin の計算より速い。クォータニオンの補間で頻繁に使うため、専用版を用意しておくとも良い。 x が 0 でも特別扱いせずに済むし、高速である。

4.5.2 絶対値、平方根、最大、最小

絶対値、平方根、最大、最小に関しては CPU によってはそのものズバリな命令を持っており、非常に高速に行える事がある。呼び出し回数も多いので、持っているなら用意しておいて損はない。

また、逆数平方根はあると便利である。ベクタの正規化のように、平方根で割るケースは意外と多い。これも CPU によっては命令を持っていることがある。ない場合でも、低精度の近似命令を持っていることは多く、この場合はニュートン法で改良してやれば良い。標準の `fsqrt()` の実装が手抜きな場合には、こうして自分で用意した方が速いことがある。ちなみに、

ニュートン法による改良は、以下ようになる。これは x86 で SSE 命令を使った場合の例だが、他の CPU でも近似値をもらった後の計算は変わらない。

```
inline float rsqrt( float x ){
    __m128 vx = _mm_set_ss( x );
    vx = _mm_rsqrt_ss( vx );
    float z;
    _mm_store_ss( &z, vx ); //11bit
    //ニュートン改良 2 回
    z = ( 0.5f * z ) * ( 3.f - ( ( x * z ) * z ) );
    z += ( 0.5f * z ) * ( 1.f - ( ( x * z ) * z ) );
    return z;
}
```

ニュートン法が 1 回目と 2 回目で書き方が異なるのは意図的である。「近似値のプログラミング」を参照すると良い。

なお、重ねて言うが、こうやって自作した方が速いということは保証しない。また、1bit の誤差もなく標準と一致することも保証しない。CPU や、標準関数の実装、コンパイラ、コンパイラオプションに依存する。インライン関数として実装しているため、使う文脈にも依存する。実際、上記の関数は他機種での実装を楽にするためのテンプレートとして PC に用意したものであり、実用のためではない。PC の場合 CPU の種類が一種類ではないため、こうした努力は裏目に出る可能性が高いし、そもそも PC 版で CPU 性能が問題になることはありえない。あったとすれば、それはゲームの設計がそもそも間違っている。PC 同様に CPU が多種にわたる携帯電話の類においても同じことが言えるだろう。完全にマシンがわかっているゲーム機でなければこの手の最適化は行う価値がなく、この手の技術の価値は日に日に落ちているように思われる。ここでは参考のために紹介しておいた。

なお、浮動小数点数の絶対値に関しては、整数として解釈して符号ビットをクリアする、普通に if 文で書く、?:演算子を使う、CPU の命令を使う、などの選択肢がある。どれが速いかは場合によるし、実際のところさしたる差はない。ただし、整数として解釈して良いのはロードヒットストアがない CPU に限られる。なお、単に標準の fabsf() を呼ぶのが一番速いというケースも結構あった。SegaLib は保守性を鑑みてよほどのことがない限り関数をまるごとアセンブラで実装することはしないようにしているため、標準に勝てないケースはどうしても出てくる。

名前空間問題

SegaLib では、sin()、cos() といった標準と同じ名前に関数を用意している。つまり、math.h をインクルードしている場所があると名前が衝突し、いちいち Sega::sin() と書かねばならなくなる。楽に使えるようにするために、また SegaLib 外のコードからの移植を容易にするた

めに同名にしたのだから、これでは目的が果たされない。したがって、`math.h` はインクルードされないことを前提としている。標準を使う場合でも、`cmath` をインクルードすれば `std` 名前空間に入るため、この問題は発生しない。

しかし、プラットフォームによってはシステムのヘッダをインクルードすると中から `math.h` がインクルードされて、`sin` や `cos` がグローバル名前空間に漏れ出しているケースがある。このような場合、アプリケーションのどこでも `Sega::sin()` と書かねばならない状況に陥り、極めて面倒である。こんなことがあるとは当初は想像もしていなかった。Sega 名前空間内にその関数を置けば、先に Sega 内の名前が検索されるので問題ないが、アプリケーションコードはグローバル名前空間内にあるので、やはり衝突する。極力アプリコードでシステムヘッダをインクルードしないようにするしかない状況である。

抜本的な解決策は見つかっていないが、現状は毎回 `Sega::` と書くか、`sine()`、`cosine()` のように名前を変えるかのどちらかしかない。

4.5.3 SIMD 命令

今時の CPU には、たいてい複数のデータに同じ計算を並列で行う命令が積まれている。これは場合によっては性能を倍以上に高めてくれるのだが、実際のところ使うことのメリットはそう大きくない。

まず、16 バイトアラインメントを要求されると扱いが面倒になる。SegaLib のメモリ管理は容量効率を高めるために 8 バイトアラインメントであり、別にアライン付き確保関数を呼ぶ必要が出てくる。

また、3 次元ベクタに使う場合は容量が増える。33% というのはそう小さなものではない。メモリが足りる足りないの問題もあるが、現代の計算機においてはメモリアクセスが相対的に高くつくのでかえて遅くなる可能性もある。

さらに、SIMD 命令の自由度は概して低い。ほしい命令が綺麗に用意されていることは稀であり、すでにあるデータを SIMD 向けに再配置せねばならないケースが多くある。再配置の負荷まで考えると採算がとれないわけだ。もちろん、最初から SIMD を考慮したデータ構造にしておけば良いように思えるが、ここで複数機種対応であることが祟る。SIMD を持たず性能が低いマシンが混ざっている時、SIMD 前提のデータ構造にするとメモリを無駄遣いしつつも性能が上がらないという辛い状態に陥る。SIMD の命令セットによっても適したデータ構造は異なる。その結果まるごと機種別に作らなくてはならない。

加えて、SIMD 命令が額面通りの性能を発揮することはそう多くない。SIMD 命令を呼ぶのに何かを切り替えねばならないケースもあり、そういうケースでは切り替えにオーバーヘッドがかかる。また、4 並列で計算する命令なのに、中身は 2 並列でしかなく、普通に `float` で計算した場合の 2 倍にしかならないケースもある。SIMD 命令に使うレジスタが別物で、通常の

整数レジスタや浮動小数点レジスタとのやりとりが非常に遅くなることもある。さらに、そもそも 2 並列の命令しか持たないハードウェアもある。そういうわけで、素人が想像するほどの効果は出ない。普通に float を使う C のコードで書いた方が速いケースは珍しくない。

そういうわけで相当がんばらないと性能は高まらないし、そのがんばりは完全に機種ごとにバラバラに行わねばならない。根本的に CPU は「同じ計算を多数のデータに行う」という用途には弱いものなのであって、CPU でがんばる暇があったらそれを得意とする別のものを使うべきだ。PS3 なら SPU があるし、GPU が使えるケースもある。

SegaLib においてはスキニングを CPU で行う機能を提供しているが、この中身は機種別に SIMD 命令を使った実装になっている。GPU でのスキニングが使えない場合や、GPU 負荷を軽減するために CPU に負荷を押し付けたい場合には、最適化による性能向上が非常に重要になる。このためにかかなりの労力をかけて高速化を行った。

4.6 メモリ管理

PowerSmash4 は C++ で書かれており、メモリは基本的には new で取る。どのマシンであれ、何もしなくても new と書けばメモリは取れる。したがって、その意味でメモリ管理に関してせねばならないことはあまりない。

しかし、致命傷となるバグのほとんどはメモリ関連である。解放し忘れていずれはメモリを確保できなくなったり、確保していないメモリに書きこんで他で使っているメモリを破壊したりするケース、さらには初期化していないメモリをそのまま使ったり、解放したメモリをもう一度解放してみたりと、致命傷となるメモリ関係のバグはいくらでも列挙できる。これらのバグを見つけるための支援機能が何もない状態でゲームのような大きなプログラムを作り上げるのは相当に難しい。今までの私の経験からすれば、不可能と言ってもいいレベルの難しさである。したがって、ライブラリには最初からこれを見越した機能を組み込んでおくことが望ましい。

さて、Windows 上で VisualStudio を使ってプログラミングしていれば、少しの手間でこのための機能が手に入る。crtdbg.h をインクルードして、若干のコードを書けば良い。メモリを解放し忘れていれば教えてくれ、さらにはどのファイルの何行目にある new で確保したメモリがまでわかる。メモリ破壊が起こればかなり早い時期に何が起こったことを知らせてくれる。このため、windows 上でプログラミングをしている限りにおいては、この手の機能を自作せねばならないわけではない。

しかし、PowerSmash4 は複数機種対応である。ゲーム機の SDK にそこまで親切な機能が埋め込まれていることは稀であり、PC 版以外は自作せねば望むものを手に入れることはできない。そして、この手のものを機種別に書くのは馬鹿馬鹿しいわけで、全機種で可能な限り同じコードを通すことが望ましい。そういうわけで、SegaLib ではライブラリの最も基本的な機

能として、全機種同じしくみでこのデバグ支援機能を提供している。

4.6.1 実装

PowerSmash3 で使っていたライブラリは、operator new の中で少し大きなサイズで malloc() を呼び、先頭にファイル名や行番号、破壊検出用の特別な番号、そして何らかの文字列情報などを埋め込んでいた。malloc() そのものはシステムのものを使えるので、実装は簡単である。しかし、同時に、何回 new が呼ばれたか、解放されていないメモリブロックはいくつあるか、などの統計情報をグローバル変数に入れており、そのためにミューテックスをロックしていた。つまり、複数スレッドから呼び出すと new がかなり重くなることがあった。そもそも、malloc() そのものが中でスレッドセーフになるような処理をしており、本質的には無駄な処理である。

さらに、デバグ情報や破壊検出用の番号などがかなりのサイズになっていたため、一回 new する度にかなり余分なメモリを消費していた。破壊検出用の領域が広がった頃には 64 バイトで、あまりに辛いということで縮小してもらった後も 16 バイト以上は消費していた。malloc() がすでに中で余計にメモリを使っているため、余計なメモリ量はさらに大きくなる。当時のライブラリが new の回数に無頓着な作りだったこともあって、new の負荷と無駄になるメモリ量が無視できない状況だったのである。今回はその轍を踏まないように心がけた。

まず、最低限 new のあるファイル名と行番号がわかれば良い、というように機能を限定した。これだけあればメモリリークを調べることができる。これ以上あったところでさして楽になるわけではない。確保回数と確保総容量は記録しているが、「目安」とした。つまり、複数スレッドが競合して書き込んだ場合には間違ふ可能性がある。どうせ目安にしか使わないのだから、ミューテックスでロックする必要などないのである。また、個別の new に名前をつける機能や、破壊検出用の番号 (マジックナンバー) なども捨てた。ただし、最終的にはビットをやりくりすることで、0 から 15 までの任意の番号を含められるように拡張されている。例えばグラフィックス、アニメーション、のような種別ごとに番号を決めておくことで、メモリの使用状況を把握しやすくなる。

さらに、malloc そのものを自作することによって、無駄になる容量を最小化することにした。それに機種共通の仕組みで作っておけば、断片化のパターンまで含めて全機種でおおよそ似た感じになるはずである。

malloc は単純なアルゴリズムで良ければ K&R にあるような基本的な実装でも動く。しかし、運良く少し調べただけで glibc の malloc について解説している文書を web で発見でき、それをそのまま実装することにした。1MB 単位で OS からメモリをもらい、ここから切り出して渡す。この時、ファイル名の番号 14 ビットと、ファイルの行番号 14 ビット、それに 0 から 15 までの好きな番号を入れる 4 ビットの計 4 バイトを含めておく。デバグ支援機能を 4 バ

イトで実現できたため、この機能はデバッグビルドだけでなく出荷用のライブラリでも有効なままにしておいた。デバッグ用ライブラリと出荷用ライブラリの違いをどのようにするかについての基本的な考え方はすでに述べた通りである。ことメモリ管理に関しては出荷用だからといって削ることはしていない。これによって出荷寸前までメモリ関係のバグを調べやすい状態を保つことができる。

なお、複数スレッドから malloc を呼んでも極力遅くならないようにする仕組みや、高速化の工夫、メモリの無駄を極限まで削る工夫については私が参考にした資料に書かれている。^{*2}

また、ファイル名が 2 バイト (14bit) で格納されているのは、グローバルの配列にファイル名文字列のポインタを入れておき、その番号だけを入れているからである。配列は開番地法のハッシュで実装されており、それなりに高速に検索できる。容量を削るために速度を犠牲にしたわけで、少々やりすぎたかもしれない。

4.6.2 new の上書き

new と書いた時に自分で書いたメモリ関数が呼ばれるようにするには、operator new() を実装すれば良い。operator new() は通常 size_t のみを取るが、他の引数があるバージョンを作ることができ、例えばファイル名と行番号を渡せるものを作る。

```
void* operator new( size_t size, const char* filename, int line ){
    return MemoryManager::instance().allocate( size, filename line );
}
```

という関数をどこかに用意しておけば、MemoryManager なるクラスの allocate 関数の戻り値のメモリにコンストラクタを呼び出した上でポインタを返す new が出来上がる。delete についても同じで、

```
void operator delete( void* p, const char*, int ){
    MemoryManager::instance().deallocate( p );
}
```

という具合である。これでデストラクタを呼んだあとで MemoryManager の deallocate にポインタが渡されるようになる。配列版もそれぞれ必要だ。あとは、毎回

```
int* a = new( __FILE__, __LINE__ ) int[ 4 ];
```

などを書くのが面倒くさいので、例えば

```
#define NEW new( __FILE__, __LINE__ )
```

^{*2} 現在資料は見当たらないが、発表のビデオがある。(http://video.google.com/videoplay?docid=2914803742593360351)

のようなマクロをどこからでも見える場所に置いておく。これで、

```
int* a = NEW int[ 4 ];
```

と書くだけでファイル名と行番号が埋め込まれるようになる。

new の上書きの手順については別途調べてほしいが、大まかには上のようなものになる。operator delete() の呼び出しが operator new() と対応していなかったり、機種によっては特別な配慮が必要だったりして何かと面倒なのだが、この手間をかけることで、開発終盤のデバグの手間が大幅に減るのでおろそかにはできない。

また、現在使用中のメモリブロックの情報をまとめて吐き出す関数や、わかる範囲でメモリ破壊の検査を行う関数なども用意してある。とあるマクロを定義すると、new や delete の度に全メモリブロックの整合性をチェックする強力デバグモードになるようにもしてある。

4.7 スレッドと同期

今時、スレッドが全く使えないマシンはそうそうない。幸い SegaLib が対象とするマシンは全てスレッドを持っている。CPU が一個しかないケースなどもあって、必ずしも高速化のために使えるとは限らないが、長時間かかる処理を空いた時間にだけやるようにする、といった処理の非同期化のためのスレッドは全機種で使えるということである。

しかしながら、スレッドやそれに必要な同期機構には、それぞれのマシンによって微妙な違いがある。

4.7.1 スレッドのスケジューリング

幸いにして、どのマシンにも優先度という概念はあり、優先度が高いほど先に実行される。ただし、優先度というものがあっても、それをどう使うかは必ずしも明らかではない。実装によっては「優先度が高いほど実行されやすい」というくらい曖昧かもしれないし、あるいは「優先度が高い処理があるうちは低い処理は一切行わない」とよりはっきりとした動作であることもありうる。

また、同じ優先度のスレッドがある時にどうするかもマシン次第である。それなりに時間を区切ってまんべんなく実行する場合もあるだろうし、一回実行が始まると終わるまで一切止まらない場合もあるだろう。

さらに、複数の CPU がある時に、あるスレッドがどの CPU で実行されるかも定かではない。スレッドが実行される CPU が固定されて空いた CPU があっても実行されないこともあるだろうし、一つのスレッドがその時に空いている CPU に動的に割り振られることもあるだろう。

そういうわけで、スレッドというものが提供されていたとしても、それがどう動くかは必ずしも明らかではない。そして、SegaLib は複数機種対応のライブラリであり、これを使うアプリケーションは一つコードを書けば複数の機種で同じように動くことを期待している。したがって、全てのコードはスレッドのスケジューリングがどのようなものであれ問題なく動くように書かれていなくてはならない。

まず、SegaLib はアプリケーション側が優先度を設定する方法を用意しない。立てたスレッドは全てメインスレッドよりも低い優先度となる。よって、リアルタイム性の高い処理はメインスレッドで行わねばならない。つまり処理落ちないように作れよ、ということである。もちろん、SegaLib 内部では必要に応じて優先度の高いスレッドを立てたりもする。あくまでアプリケーション側からは優先度を設定できない、ということだ。

次に、全てのスレッドは一旦実行が始まったら、自発的に寝るか、return で処理を終えるか以外に実行を止める方法はない、という前提で書いてもらうこととし、SegaLib の内部でもそうしている。仕事がなくなると自発的に寝るようになっている。これをもし、優先度が低いことをいいことにひたすらぐるぐるループし続けるような処理にすると、スケジューリングのやり方次第では他のスレッドに永久に処理が戻ってこないことになりかねない。

なお、基本的に使用者に対してはスレッド関連の機能は最小限しか提供していない。

SegaLib::Thread クラスを継承したクラスを作り、operator()() に処理を書き、start() を呼ぶと始まり、wait() を呼ぶと終了を待つ。これだけである。スレッドの強制終了は許さない。終了したかどうかを問い合わせる関数もない。単に wait() で待つ以外のことはできなくしてある。wait() を呼ぶ前にデストラクトすると ASSERT で止まる。

```
class Thread{
public:
    Thread();
    virtual ~Thread();
    void start();
    void wait();
    virtual void operator()();
};
```

基本的に、SegaLib としては使用者が自分でスレッドを立てることを推奨していないことはスレッドプールの所にした通りである。

4.7.2 スレッドプール

すでに述べた通り SegaLib ではスレッドプールを提供している。これは、スレッドを使っ
て欲しくないからである。

スレッドの使い方は大きく分ければ二種類あり、一つは非同期化、一つは並列化である。

非同期化とは、長い処理を裏に回したり、何らかのサービスを起動した状態にするためである。これはスレッドを立てなくてはできないことだが、音声、ファイル読み込み、グラフィックス、パッド入力といったあたりをライブラリ内部で持っていることもあって、アプリケーション側でやる頻度はそれほど高くない。機種固有の機能や、セーブデータの書き込み、OSの持つダイアログやネットワークなどで必要になる程度であり、ゲームの中身の処理に使うことはまずない。その手の単機能スレッドは性能に影響は与えず、バグを入れる危険もそう大きくはない。

一方並列化は高速化のためであり、ゲームの処理がシングルスレッドで手に余るようであれば積極的に利用していく必要がある。そして、スレッドを使って実装した時にバグの原因になるのはこのような使い方のスレッドである。その害を少しでも軽減するため、スレッドを立てるという手続きを省略し、またできる処理の種類を限定するためにスレッドプールを用意することにした。

4.7.3 同期

SegaLib として必要とする同期機構は、Windows で言うところのミューテックス、イベント、セマフォ、の三つである。ミューテックスは何かと使うし、イベントは終了待ちと終了検出に使い、セマフォは非同期な仕事をキューに溜めて実行する際に、仕事がない時には寝て待つようにするために使う。セマフォの値をキューの要素数と一致させておけば、0の状態からキューから仕事をとりようするとセマフォが0なので1以上になるまで寝ることになる。仕事がある限りは仕事をし、なくなったら仕事があるまで寝る、ということを実際に表現できる。数ミリ秒寝ては仕事があるかチェックする、というような実装だと必要以上に寝てしまって仕事を始めるのが遅くなることもあるからである。ただし、これは理屈の上でのことで、実際どの程度性能が変わるかは機種にもより定かではない。

なお、すでに述べたようにアプリケーション側では極力スレッドや同期をしないようにしてもらっているため、これらを使うのはほぼ SegaLib 内部だけである。スレッドプールに投げるジョブは極力同期を必要としない単純なものにもらっているからだ。

なお、ここでイベントとセマフォが問題になる。ミューテックスが提供されていないマシンは現状ない。しかし、イベントとセマフォは存在しない場合がある。ひどい場合になると、ヘッダにはあるのに実装がなくて動作しないケースまである。イベントやセマフォがないマシンでは条件変数が提供されているので、条件変数とミューテックスを使ってこれらを自作することになるのだが、条件変数は一歩扱いを間違えるとスレッドが永遠の眠りについてしまうために非常に厄介である。ただし、一度できてしまえばどの機種でもほぼ同じようなコードで書けるため、全体から見ればかかる手間はさして大きくはない。また、自作したセマフォが

正式に提供されたセマフォよりも性能が良ければ、自作してしまうのも手であろう。例えば Windows のセマフォはプロセス間の同期に使えるがゲームでは不要であり、もしかしたら自作することで性能が上がるのかもしれない。もちろん、実際に試したわけではなく、可能性の話である。

なお、コンペアアンドスワップなどのより低レベルな機能を使って同期することも可能で、場合によってはその方が高速だが、現代のゲーム開発においては消費電力の低減も重要であり、いくら性能が高いからといって条件が満たされるまでひたすら while でループするようなプログラムは歓迎されない。デッドロックの危険も高まる。そのため SegaLib では「同期に引っかかったらスレッドは寝る」ということを基本にしている。

最後に、オーバーヘッドについて述べておく。同期のオーバーヘッドはマシンによって異なる。スレッドを寝かせたり起こしたりすることがべらぼうに高くつくマシンもあるだろうし、そうでもないマシンもあるだろう。

スレッドプールにジョブを投げる事を考えると、ジョブを格納するキューはミュutex で保護され、さらにセマフォで実行スレッドとの同期をとっている。各ジョブの終了待ちにはイベントを使っている。そういうわけで、ジョブの数に比例して同期機能の使用回数が増える。下手に並列化することでこれらのオーバーヘッドがかさんで、余計に遅くなる可能性がある。よくよく注意せねばならない。理屈の上では細かいジョブほど良さそうに見えても、実際には最適なジョブの大きさはマシンによって異なる。ジョブはかなり大きな塊にしておき、数を抑えるのが無難であろう。ジョブをいくら増やしても、CPU の数しか同時には実行できないのである。

4.8 ファイル IO

SegaLib がサポートするのは読み込み専用ストレージからの読み込みだけである。セーブデータの類は機種ごとにあまりにも使い勝手が異なるため、サポートは諦めた。アプリケーション側がセーブやロードを作り始める頃には私はもうアプリケーションのコードを書かねばならない時期になっており、そこまで面倒を見る余裕がなかったというのものもある。したがって、SegaLib のファイル IO 機能は、開発時に作ったデータをロードすることに特化している。しかし、それでも事はそう簡単ではない。

4.8.1 機種ごとの差異

ファイル IO は、つまるところ `fopen`, `fread`, `fseek`, `fclose` があれば理屈の上では事足りる。これらの関数が各機種で提供されていれば、機種固有コードを書く必要はなくなる。しかし、残念ながらそうとは限らない。

C 標準ライブラリでのファイル読み込みができない場合、当然 SDK がファイル IO の機能を提供しているのだが、ここには機種ごとの差異がある。

まずはそれが同期アクセスか非同期アクセスかだ。同期アクセスの場合、読み出しが終わるまでスレッドが止まるので、メインスレッドからは呼べなくなるし、エラーが起こった時に無限リトライするような仕様だとエラー監視スレッドが別に必要になって厄介である。

もう一つはアラインメント制限である。一定単位でしか読めなかったり、データを書きこむバッファのアドレスのアラインメントに制限があったり、ファイルの読み出し位置にアラインメント制限があったりする。これが機種によって異なるということだ。

そして、最大の問題がエラー処理である。お客さんがディスクを抜いた時に何が起るかはマシンによる。勝手にゲームが終了する場合もあれば、抜かれたことを検出して「抜かれてますよ」と画面に出さねばならないこともある。また、ディスクを読みそこねた場合に、勝手にもう一度読みに行く場合もあれば、エラーを返して諦める場合もある。もう一度読みに行く場合も、永遠に挑戦し続ける場合もあれば、それなりに繰り返した後諦める場合もある。

ここの対応は非常に厄介だ。

後に述べるように、ファイル IO はスレッドを用いたかなり複雑な処理である。エラーが起きた際に矛盾なく処理するのは結構面倒だし、極力機種依存のコードを書かずに処理しないとコードの保守性が悪化しすぎる。なお悪いことに音声や動画のストリーミング処理が絡むと事はさらに複雑化し、SegaLib 内でも屈指の複雑な部分になっている。正直もうやりたくない。しかも、そういった問題が発覚するのは、ゲームがおおむね出来てテストを始めた頃なのだ。

4.8.2 クラスの構造

SegaLib では、使用者から見えない所に非同期ファイルアクセス風のインターフェイスを持つクラスを用意してある。ここでは File と呼ぼう。開け、読み出し要求をし、読み出し終了を待ち、閉じる。

```
class File{
    void open( const char* filename, ... );
    int size();
    void read( ... );
    void wait( ... );
    void close();
};
```

また、アラインメント制約の値を調べる機能も持つ。このクラスを使う場合にはアラインメント制約に従わなくてははいけない。例えば PC 版では、OS のファイルキャッシュの状況でロード時間が変わらないよう非同期の非バッファアクセスを基本にしており、この場合 512 バ

イト単位でしか読み出せない。要求サイズが 512 の倍数でなければ ASSERT で停止する。なお、機種によってはこの中が fopen や fread で書かれていることもあり、この場合は読み出し要求を出した段階で読み出しの完了を待ってしまう。読み出し待ちの関数の中身は空だ。インターフェイスが非同期的なだけで、実際の処理が同期か非同期かはわからない。これが一番下の層であり、使用者には公開していない。

この上に、ファイル IO を非同期化するクラスがある。上層からファイルの読み出し要求を受けて、これを下の File に流す。このクラスはスレッドを立てて、キューにある要求を順次処理する。ここでスレッドを使っているため、File が同期か非同期かは問題ではないわけだ。また、このクラスはエラー処理も行っており、ディスクが抜かれていたり、プログラムが終了する間際だったり、ディスクが汚くて読めなかったりした時に対処する。すべき対処は機種によって異なったりもするのだが、できる範囲で共通化してある。ここには後述の FileManager が要求を投げてくるが、音声再生や動画再生からストリーミングアクセスの要求が来ることもある。そして、このクラスは使用者には公開していない。

さらに上に、使用者から見える FileManager というクラスがある。FileManager はファイルをまるごと読むためのクラスである。使用者は FileManager に「このファイル名のものを読んで」と頼む。使用者ができることは、その後ロードが終わったか聞くことだけだ。SegaLib は複数ファイルを 1 ファイルにまとめるツールを提供しており、場合によってはまとめたファイルの中から要求されたファイルを出してくる。また、この場合は圧縮がかかっていることもあり、必要に応じて展開する。FileManager もスレッドを立てており、ファイルを開けたり、圧縮されたものを展開したりする場合にメインスレッドが止まらないようにしている。また、同じファイルに対する要求が来た場合は一回しか読まないように重複の除去も行っている。それぞれのファイルのデータは参照カウントで管理される。このため、読んだデータは読み込み限定である。読んだデータを修正して使うような使い方は行儀がいいとは言えないので、私としてはやめていただきたい。どうしても必要な場合は別にバッファを用意してコピーしてもらうことになる。

4.8.3 ファイルまとめ

ファイルを開ける処理は、マシンによっては高くつく。というよりも、それが重くないマシンを探す方が難しい。DVD にせよ HDD にせよ、ファイルを開ける処理は CPU 処理的にも機械の動作的にも時間がかかる。また、ファイルがたくさんあればどうしてもアクセスは遅くなるし、配布の際にも一個だけファイルが抜けたりして問題が起こりやすい。さらに、ファイルには圧縮をかけたくなるものだが、ファイル一つ一つに圧縮を施すのは面倒である。

そこで、SegaLib では指定のディレクトリ以下をまとめて 1 ファイルにするツールを用意している。結合は「同じディレクトリにあるものが辞書順に連続する」ことを保証する。例えば

「1面」というディレクトリに1面用のデータを入れておいてまとめてロードすれば、おおむね連続して処理され、ロード時間が短くなる。明示的に複数のファイルを連続して配置し、読み出しを一括で行うようにも設定できる。また、圧縮を試みて小さくなるようであれば、圧縮して格納される。展開はFileManagerによって自動で行われるため、使用者は気にしなくて良い。展開はロードと並列化されるため、よほどCPUが忙しくない限り、展開によってロードが遅くなることはない設計である。ただし実際にそうなっているかは測定していない。

なお、まとめたファイルは2GBが上限である。2GBを超えると、自動でファイルを分割する。これは、マシンによってはファイルサイズに限界があるケースがあるからだ。後に述べるように、SegaLibは基本的には機種が異なっても同じファイルを使うことを推奨している。実際PowerSmash4は360版とPS3は同じまとめファイルをロードしている。そういうこともあり、どの機種でも読めるファイルサイズということで2GBを上限としておいた。これはそのうち変わる可能性もあるが、半導体メモリで配布したり、ダウンロード販売したりするならば容量は増やせないわけで、現在の傾向を見るに2GBから増やす必要があるとは思えない。仮に20GBくらいの大きなゲームを作ったとしても、ファイルが10個になるだけで、さしたる問題ではないからだ。いくつに割っても使用者が書くコードは同じである。

なお、ファイルまとめはかなり時間のかかる処理である。数千ファイル以上で合計がギガ量になるようなものを読んで圧縮して書き出すわけで、分のオーダーとなる。とりわけ開発終盤には頻繁にこの作業を行うため、実行時間がストレスになる。というわけで、前述のスレッドプールを使ってファイルごとにジョブを立て、圧縮処理を並列化している。ただし、結局一番遅いのはファイルアクセスであり、開発PCのストレージがSSDにでもならない限り劇的な向上は見込めない。SSDの普及が望まれるところである。

4.9 標準ライブラリや言語機能について

C++言語は大きな言語である。C標準ライブラリに加えてC++標準ライブラリを持ち、言語としても例外やRTTI、多重継承など豪華機能が目白押しである。先進的なプログラムは新しい機能は積極的に使うべきだと言うし、標準ライブラリは性能的にも機能的にも良く練られており、また、多くの人が習得しているものであるから積極的に使うべきだと言う。

一理ある。

しかし結論から言えば、SegaLibの中ではこれらをほぼ使っていない。C標準ライブラリで使っているのはmemset, memcpy程度であり、これも機種によってはSDKが提供する高速版に差し替えている。C++標準ライブラリに関してはほぼ使っていない。vector, list, map, stringなどの当たり前に使われる機能も使っていない。例外はPC版のみツール及びデバッグ向けに一部使用しているが、RTTIはなく、多重継承もない。そもそも継承や仮想関数からして一般的なC++プログラムと比べて極めて少ない。

ここではこのようにした理由について触れ、代替物の設計と実装についても触れる。

4.9.1 何故標準が嫌なのか

ゲーム機が無限の性能を持ち、全プログラマが C++ の隅から隅まで熟知しているならば、これら機能を使わない理由はない。しかし、現実にはいずれの条件も満たされないため、必ずしも標準を使うことが妥当であるとは限らない。もちろん、ゲーム機が十分な性能を持ち、大多数のプログラマが C++ をある程度以上知っているならば、これらの機能を使うことは妥当である。しかしこのように条件を緩めてみても、なお不安は尽きない。

まず C++ 標準ライブラリに関してだが、これは安全性と性能の両面で危険な面がある。例えば `vector` には `push_back()` や `erase()` がある。前者が中で `new` を呼び出す可能性があり、後者が多数のデータのコピーを行いうることを皆が理解していれば良いが、そうではない。もし皆が理解しているならば `erase()` など使うはずがなく、したがっていずれにしても `erase()` は必要ない。削除で使うのは `pop_back()` だけである。

`list` にしても、`insert()` を行えば普通は `new` が走る。実装によっては 2 回走る。アロケータを自前定義して高速な `new` を提供すれば良い、という方針を私は採らない。私は `new` や `malloc` については極めて保守的であり、ゲーム実行中に毎フレーム行う `new` があることを許したくない。現実には 0 というわけには行かないことは承知しているが、`vector` や `list` がそこから中で使われている状況では、0 どころか、10 や 100 ですら達成は難しくなる。

整理すれば、使うべきでない機能が数多くあること、実装によって挙動が異なること、そもそも `new` が多数走るようなものは使いたくないこと、という 3 点により、STL は使いたくない。SegaLib 内部にはないし、アプリケーション側のプログラマにも事あるごとに排除するように勧めている。

無邪気な STL の使用がどういった害を与えるかに関して、ひとつだけ例を挙げておこう。

```
std::map< string, Model* > modelMap;  
Model* model = modelMap[ "hero" ];
```

名前をキーにして、モデルクラスのポインタを取得する。この手の検索処理はよく行われるが、このコードで `new` が走ることはなかなか気づかれない。`map` の `operator[]` に渡されるのは、`"hero"` という `const char*` でなく、そこから生成された `std::string` である。そして `string` のコンストラクタでは `new` が走る。よって、この検索では毎回 `new` が走るのである。検索そのものが遅いことが理解されていて、毎フレームやるような書き方をしていなければそれでもいいが、「`map` の検索は速い」と簡単に理解されているケースは多い。実際これは PowerSmash4 の開発中に問題になったことである。

なるほど、現代のコンピュータは十分に高速であるとか、多少速度を犠牲にしても生産性を

重視すべきであるとか、80:20の法則の通り大半のコードは遅くても良いとか、そういった意見はある。どれも一理ある。しかし、常にそうだというわけではない。ゲーム機の性能は必ずしも十分ではない。よほどアイディアが優れていない限り、機械の性能を引き出さなくて良いという決断はなかなか下されない。とりわけ成熟著しい家庭用据え置きゲーム機市場においてはそうである。そして、携帯機は性能が厳しく、生ぬるいコードを書けばあっという間に処理落ちする。

なるほど生産性は重要で、一刻も早く動く実装を用意することは重要である。私とて、本当に速度がどうでもいい場所であれば、あるいは後で直すべきことをメモした上であれば標準のものをありがたく使わせていただく。しかし、多くの場合こうした実装は無意識になされ、開発終盤に速度が不足して時間のかかる性能調査をするまでは問題が発覚しない。すなわち、時間短縮どころか、問題を調べる作業と、それを修正する時間が、ただでも時間がない開発末期に襲ってくることになる。

80:20の法則は確かに成り立つが、ゲームのコードが10万行あれば、20%といえども2万行になる。漫然と眺めていては日が暮れる量だ。まして、最近の大きなゲームは50万行を超えることも珍しくない。PowerSmash4はさして大きなゲームではないが、50万行は超えている。結局0.1%の負荷がかかる部分が1000箇所ある、というような状況になり、「最適化せねばならないごく一部の部分」がすでにして膨大な量になる。ただでも忙しい開発末期にそんな不毛な作業はしたくない。問題はより上流で解決するほど安く済む、という一般的な傾向を思い返すべきである。日々の習慣として遅くないコードを書く方が良い。それによって余計にかかる時間など高が知れている。

そういう考えで、多少不便ではあるが、遅い書き方をしにくいように設計したSTLの代替物をSegaLibに用意しておいた。

なお、このような私の考えが社内で優勢なわけではない。むしろ少数派に属する。大抵のプログラマは自分のスキルに自信を持っており、また、他人のスキルも信用している。そして、プログラマたるもの新技术には敏感であるべきであり、STLやBoostを使いこなせないのは恥であると考える者も多い。こういった考えの人は、保守的な私の考えをとりわけ強く嫌う。加えて、「お前の作ったものが標準のSTLやBoostより信頼性があると信じる理由はない」と実にもっともな意見もある。

私も、このやり方が完全に正しいと言う気はない。だから、STLやBoostを併用することを邪魔はしないし、実際されている。ただし、SegaLibの関数のいくつかはSegaLib独自コンテナを引数に取るため、強制的に使わざるを得ない場合はある。もちろんこれは意図的にやっている。

4.9.2 用意したもの

以下に STL や C 標準ライブラリにあるようなものを自作した例を紹介する。ただし前もって申し上げておくが、「作ってみたかった」という動機がかなり大きかったことは確かである。したがって、本当に性能上の優位があることをすべてにおいて確認したわけではない。

vector, list, hash_map, hash_set, stack, queue に似たもの

基本的に new の排除あるいは軽減が目的であり、メモリを連続的に使ってキャッシュヒット率を上げることもおまけとしてついてくる。最初に容量を決めたらそれ以上には拡張できないものと、一定個数ごとにまとめて確保して拡張できるバージョンがある。また、遅い処理は提供していない。例えば vector もどきや list もどきには検索がない。vector もどきには末尾以外の削除がない。

iostream, printf

デバッグ用には printf や cout などの文字列出力機能が必要になる。しかし、これを直接使うと後々面倒の元になる。

まず、この手の機能は出荷時には削除しなくてはならない。しかし、標準関数を直接呼んでいる場合、使用箇所をすべて削除せねばならず、手間もかかるし、残ってしまう危険もある。そこで、SegaLib では別にデバッグ文字列表示用クラスを用意し、その型のグローバル変数として cout という名前のもを用意した。出荷用の設定にすると何もしない実装に切り替わるため、アプリケーション側のコードは変更せずに済む。また、機械によっては printf や cout の出力がデバッグ出力に出ないケースもある。これも自前で用意することで、デバッグに出力させられる。

なお、printf や sprintf などの可変長引数で指定するやり方に慣れたベテラン世代が多く、C++ 風のインターフェイスしか用意しなかったことには苦情が多く出た。彼らの printf への愛着は予想外に強く、アプリケーション側で可変長引数を取る関数を上にかぶせられてしまったほどである。社内には未だに C 言語でゲームを作っているチームもあるようで、ベテラン層というのはそういうものなのかもしれない。しかし、いずれ時間が解決してくれるだろう。

sprintf や stringstream

数値から文字列への変換には sprintf や ostreamstream を使うのが常だが、sprintf は中でバッファを確保することも、サイズをチェックすることもできず危険である。安全性を高めるために引数を増やしたものが提供されているケースもあるが、前機種で提供されていることは保証されない。また、デバッグ文字列出力に printf 風でなく iostream 風のインターフェイスを

採用したことからわかるように、私は C 風よりは C++ 風の方が安全性の点から望ましいと考えている。そもそも可変数引数は嫌いだし、void*を取る関数は極力減らすべきだと考えている。その意味では、ostringstream を使うことが望ましい。

しかし、stringstream には性能が恐ろしく悪いという致命的な弱点がある。何かする度の中で new が走る。文字列から数値への変換を行う istringstream も、同様に new を多発させ許容できないオーバーヘッドを発生させる。これに関しては atof や atoi を使うべきだが、これらもやはりバッファ範囲を超えて読みだしてしまう危険がある。

そこで、これも自作してみた。固定サイズのバッファを渡す ostringstream であり、サイズを超えると停止する。動的に拡張するバージョンも合わせて用意した。istringstream についても、文字列を参照で受け取り中で new しないものを用意した。これらに関しては機能は少ないものの標準よりもはるかに高速なものが作れており、安全性と速度をある程度は両立できている。

文字列

C++ の string の問題は、何をしても new が走ることである。なるほど安全性と利便性を考えればそうするしかないのだが、現実には性能も大切である。

そこで、SegaLib には二種類の独自実装を設けておいた。一つは、const char*のラッパーである。元の文字列がメモリ内に存在しており、変更を行わない、という制約下ではあるが、==で比較できたり、検索できたりする。

もう一つは string 同様に中でメモリを確保する文字列である。ただし、+=のように中でメモリ確保が走りうる関数は削ってある。拡張はできない。また、部分文字列を切り出して新しいメモリを確保したりもしない。それでも const char*しかないよりはよほど便利である。参照モードでコンストラクトすることもでき、先程挙げた map での検索の際の new も除くことができる。

```
std::map< Sega::String, Model* > modelMap;  
Model* model = modelMap[ Sega::String( "hero", true ) ];
```

と、Sega::String のコンストラクタの第二引数に true を渡すと参照モードになり、new しない。

assert について

assert をどう使うかに関しては、開発者の間でも意見は割れている。

デバッグ時にはできるだけたくさん入れておくべきだという意見もあるが、入れすぎてデバッグビルドの実行速度がどうしてもなく低下し、開発効率が耐えがたいほど落ちた例もある。

また、出荷時にどうするかの問題がある。普通はリリースビルドになれば勝手に消えるのだ

が、それは C 標準ライブラリにある `assert` の場合だ、自作するならば、消えるようにすることもできるし、消えないようにすることもできる。出荷時に `assert` のような即死コードを残しておくべきではないという意見は良く見るが、実際には運用上の問題がある。`assert` はテスト中は有効であることが望ましい。間違いがあればわかるからだ。直せる限りバグは直してから出荷すべきである。しかし、`assert` が有効なコードと無効なコードは別物であるから、`assert` を無効にして出荷するならば、`assert` を無効にしてテストしなければならない。なぜなら、ゲーム機向けのコンパイラはさほどテストされておらず、とりわけ発売間もないゲーム機ではコンパイラはたいがいバグ持ちだからである。`assert` には `if` 文が含まれるが、コンパイラの最適化バグがこの有無によって異なるコードを吐くことがあり、油断できない。そういう事情もあって、`assert` が有効化されたまま出荷してしまいたいという動機が存在する。

結論から言えば、`SegaLib` は `assert` を二種類持つ。一つはデバグビルドでだけ有効で、もう一つは出荷版でも有効である。ただし、別にマクロを定義することで、`assert` 内での即死関数を無効化することはできるため、実質出荷時には無効にすることもできる。

基本的に、デバグビルドでだけ有効な `assert` は入れられるだけ入れるべきである。多少遅くなるくらいで済むなら、バグが見つかる方が良い。すでに述べたように、デバグビルドのフレームレートはリリースビルドの $1/3$ より高ければ良いと私は考えている。 $1/2$ あればなお良いが、そこまでは言わない。それにたかが `if` 文であり、よほどの数でない限りそれほどの影響はない。ただし、性能が $1/3$ 以下になるようなデバグ機能の挿入は全体的な作業効率に悪影響を与える。あまりに回転数の多いループであれば多少は考慮した方が良い。

一方リリースビルドにおいては毎フレーム 100 回以上通るような場所に `assert` は書きたくない。しかし、リリースビルドという名前であっても、普通にゲームをテストプレイする時には使うわけで、かなり初期から使うものだ。したがってデバグ機能は可能な限り損ないたくない。そこで、毎フレーム 10 回未満しか通らないであろうと思われる部分についてはリリースビルドでも有効な種類の `assert` を使うことを推奨している。`SegaLib` の中ではそのように使い分けている。

そして、出荷時に `assert` 内の即死関数を無効にするかどうかはアプリ側の判断に任せている。`if` 文そのものは消えず、即死関数内の即死コードだけがスキップされるので、コンパイラの最適化バグによる影響は最小に保たれる。私自身は「`assert` で検出するようなバグは直してから出荷されるのだから、即死コードが入ったまま出荷しても問題ない」という立場だ。実際問題 `assert` に引っかかるような状況ではそこを通ったとしても遠からず異常終了するのはほぼ確実である。しかし、これに関してはさまざまな意見があり議論で決着がつくとも思えないので、無効化する方法を提供しておいた。

4.10 安全性について

例えばテクスチャがあるとする。テクスチャを生成する関数を呼ぶと、テクスチャクラスのポインタが帰ってくるのが普通だ。しかし、ポインタは上書きすればリソースが漏れるし、二度開放すればメモリ破壊が起こる。スマートポインタが現代的な解決策だが、コードが長くなって鬱陶しい。

さらに、GPU が絡んでくるデータの場合は後述するように CPU 側でいらなくなってもまだ GPU 側が使っていて、即座に delete できないケースがある。

そこで、そもそもポインタを返さないことにし、pImpl イディオムを活用してクラスそのものをスマートポインタにしてしまうことにした。

```
class Texture{
public:
    static Texture create( ... );
private:
    class Impl;
    Impl* mImpl;
};
```

create はポインタでなく Texture 型そのものを返す。実体は中にある Texture::Impl で、コピーコンストラクタや代入、デストラクタなどが呼ばれると、Impl の参照カウンタを増やしたり減らしたりして、中で勝手に管理する。Texture であれ VertexBuffer であれ Shader であれ、中の Impl は参照カウンタを持っており、0 になると適切なタイミングで破棄される。例えばテクスチャであれば 2,3 フレーム後に破棄される。

伝統的なプログラミングに慣れている人にとっては、ユーザ定義型の実体を現物コピーすることに抵抗を感じることも多いと思うが、STL を使っている人は iterator で頻繁に同じことをしており、実質的な差はない。しかも、この手のクラスはすべてポインター個しかメンバ変数を持たないので、ダメージはそれほど大きくない。

ただし、関数に渡したりする時にいちいちコンストラクタやデストラクタが走って参照カウンタの処理をするためにオーバーヘッドは大きくなる。なので、できれば参照渡しすることが望ましい。

```
void setTexture( Texture );
void setTexture( Texture& );
```

後者の方が高速である可能性が高い。ただし、一時変数を渡せなくなるなどの制約はある。そもそも、このオーバーヘッドが問題になるほど多数生成し、頻繁に使うクラスについては

このような手法を使うべきでない。実際、3次元ベクタクラスなどは普通のクラスである。

4.10.1 破棄について

破棄についてはもう少し詳しく述べておく。

描画に使うデータは、GPUに描画を要請してから、しばらくの間とっておかねばならない。DirectX風と言えば、setTexture()して、それを使ったdraw()をした後、しばらくの間はTextureを破棄せずに持っておかねばならない。

まず原則として、何の同期機構も入れなければ永遠に安全とは言えない、ということは確認しておく必要がある。CPUから投げた描画要求が、いつGPU側で完了するかは調べてみないとわからない。調べる、ということがつまり同期である。完了前に素材を破棄すれば、良くても絵がおかしくなり、悪ければGPUが暴走してハングアップする。DirectXやOpenGLの場合、テクスチャを破棄すれば中で勝手に程良いタイミングで破棄してくれるのだが、ゲーム機のSDKの場合はそんな親切な機構にはなっていないケースが大半だ。したがって、GPUがそれを使い終わっていることは何らかの手段で確認せねばならない。

そして厄介なことに、どれくらいで使い終わるのかは機械によっても状況によっても違う。CPU側で要求を溜めておく量や、GPUの処理手順などによって、どれくらい待ってやるのが性能的に最適かが違うからだ。よって、アプリケーション側で自発的に「使ってたテクスチャの破棄は2フレーム待つ」というようなコードを書けばいいというものでもない。昔のライブラリはそういう作りだったが、それは単一機種しか相手をしていなかったからだ。ここは何としてもライブラリ内部で面倒を見ねばならない。SegaLibでは、「フレームの始まり」のタイミングである関数を呼び、過去の描画が終わったことを確認している。何フレーム前の描画の完了を待つかは機械と設定による。

さて、以前作ったライブラリでは、Textureクラスはポインタで返していた。ただし、スマートポインタクラスに入れて返し、destroy()という破棄関数を別に設けて、deleteできないようにした。そして、destroy()の中で破棄予定リストに詰め、適切なタイミングで実際の破棄を行った。しくみとしてはそれでよかったのだが、スマートポインタクラスのせいでコードが長くなったし、そもそもdestroy()が必要な段階で面倒だった。そこで、今回はそれを一歩進めて、そもそもポインタを返さないことにしたわけである。

```
{
    Texture t = Texture::create( ... );
}
```

と書けば、ブロック出口でデストラクタが走り、またこのテクスチャは使われていないため、GPUを待たずに破棄される。一方、

```
{
    Texture t = Texture::create( ... );
    GraphicsManager::setTexture( t );
    GraphicsManager::draw( ... );
}
```

という具合にテクスチャをセットして描画すれば、このテクスチャは破棄予定リストに入り、2あるいは3フレーム後に破棄される。グローバル変数として Texture を作って入れっぱなしにする、というような事をしない限り開放漏れは起こりにくくなり、二重開放の問題もなくなる。

ただ、メモリ使用量がいつ変化するかがアプリケーションにわかりにくいという欠点はある。メモリが十分にあればいいが、そうでない場合はテクスチャを破棄した後数フレーム待つから新しいテクスチャのロードをしないと、古いものと新しいものがメモリ内で重複する時期ができてしまい、そこでメモリがあふれる危険がある。断片化の問題に対処する上でも、定期的にメモリをまっさらにすることは有効だ。そういうわけで、アプリケーション側で一切中身を考えなくていいというわけでもない。しかし、それは土台不可能な話であり、基本的な手法を組み合わせてやれる範囲で済ませるならば、このあたりで良からう。

4.11 XML パーサ

ゲームでは絵素材でもなく、音素材でもないようなファイルがたくさんできる。例えば、テニスコートごとの照明設定はテキストファイルにかかれており、これを書き換えてリロードすることでプログラムの中に数値を書き込むよりも楽に調整できるようにしている。しかし、以前はこれを担当プログラマが勝手なフォーマットをでっちあげてやるのが非常に多かった。個々に解釈プログラムを書いていたわけだが、こんなものは初心者を除けば訓練になるほど困難なものでもなく、単に面倒くさいだけだ。しかも文字列処理というものは、適当に書くと案外バグが出やすく、とりわけエラー処理ミスとメモリ破壊の温床になる。そこで、別段の理由がなければ XML を使ってもらうことにしたわけである。そうすれば、GUI でデータを設定するツールが使えたりもするし、プログラムは一つで済む。なんといっても XML はもはや現代社会の基幹技術とも呼びたくなるほどメジャーであり、扱うプログラムはタダかそれに近い価格で簡単に手に入るはずだ。

しかし、そうは問屋が卸さなかった。

ライセンス？コスト？リスク？さすがに大企業であり、そのへんはきちんとしている。そして、きちんとしているが故に、面倒くさい。結局自作してしまうことにした。以前簡単なものを書いたこともあり、そう手間でもなからうと思ったわけである。

そういうわけで SegaLib は非常に単純な XML パーサを持っている。といっても XML パー

サに必要なとされる仕様を満たしていないので、XML パーサというのは不適切だろう。XML 的なテキストファイルを解釈できる何かである。

自作の最大の利点は、コードサイズが小さいことだ。1000 行程度しかない。エラー処理は最低限だが、XML を人間が手で書くようなことをしなければ問題は無い。スキームのような大きな機能はまるまる省き、文字参照もない。わからない要素が出てきたら無視すればいい。< の次に!があったらコメントとみなしている。また、SAX もなく、DOM だけなので実装も簡単だ。そして、もう一つ、最初は予想していなかった利点もある。

4.11.1 バイナリ化

XML はテキストである。テキストである利点は人が読めることだが、解釈に時間がかかり、エラー処理が面倒で、容量が大きいという欠点がある。もし、これらの欠点を消せるバイナリ形式を定義して、そこから元のテキストに情報を失わずに戻せるとしたらどうだろう。普段はテキストで書いておき、内容がおおむね固定できたかなと思ったらバイナリ形式に変換しておく。そうすれば、実機側での負担は少なくなるし、ロムの容量も少なくなる。もし、バイナリに変換してしまった後にまた調整したくなければ、テキストに逆変換すればいい。

テキスト形式の XML を解釈する場合、必要なメモリサイズは前もってはわからない。そのため、どうしても new の回数が増える。大きな単位で取ってくれば回数は減らせるが、今度はメモリが無駄になる。数 KB の小さなものばかりであればいいが、以前関わったプロジェクトではメガに手が届きそうな規模のファイルもあった。当時の XML ライブラリは文字列一つごとに new していたため、そのアプリの数十万回の new の大多数が XML 経由だったことが発覚して焦ったことがある。今回はその轍を踏むわけには行かない。

バイナリファイルに変換する場合、解釈後に必要なメモリ量を前もって計算して中に埋めておくことができる。ただし、例えば Element とか Attribute といった内部で使うクラスが実機において何バイト使うかは、ポインタのサイズ、アラインメント制約、コンパイラの都合によって変わってしまう。そこで、実際には容量そのものではなく、クラスごとの数と文字列の総量を入れてある。ランタイムではこれらの数から必要な容量を計算して一括で確保し、順次切り出す。つまり、new は一回で済む。

そして、バイナリであれば、各エレメントに何個子がいるか、何個アトリビュートがあるかなども数える必要がないように埋め込んでおくことができる。さらに、< を探したりする手間もない。解釈は圧倒的に高速である。バイナリ化の時に文字列の辞書を作って、重複を排除することで容量も減る。加えて LZ77 で圧縮もかけている。これを逐次展開しながら読みだして解釈する。まるごと展開しないので余計なメモリも必要ない。

結果的には、巨大な XML ファイルなどなく、そこまでする価値はなかったのだが、バイナリ化によって解釈が高速化したことには意味があった。また、テキストがそのまま入っている

よりは解析への耐性も高い。

4.11.2 専用アロケータ

バイナリ化装置は作ったが、テキストの XML もまた多く、それらの読み込みも速いに越したことはない。また、メモリの無駄も少ない方がよい。そこで、まとめてメモリを確保し、そこから切り出して使うためのクラスを用意した。

```
class MemoryPool{
public:
    explicit MemoryPool(
        int defaultBlockSize = 1024,
        int alignSize = 8 );
    ~MemoryPool();
    ///n バイト確保
    void* allocate( int n );
};
```

allocate を呼ぶ度に中のメモリを切り出し、足りなくなると一定のサイズでまとめて確保する。例えば 1024 バイトづつだ。これでメモリをもらって、そこに placement new でコンストラクトすれば良い。

```
T* t = memoryPool.allocate( sizeof( T ) );
new( t ) T( ... );
```

なお、解放関数はない。XML などのファイルからのデータ構築の際には途中で解放が必要になることはそうそうなく、稀にあったとしても解放しないで放っておけば良い。解放は MemoryPool のデストラクトでまとめて行われる。確保したクラスのデストラクトは手動で行えば良い。

```
t->~T();
```

途中で解放さえしなければ、メモリの切り出しの実装は簡単である。単に「ここまで使いましたポインタ」をサイズ分インクリメントすれば良い。アラインメントにさえ気をつければいいだけである。

この MemoryPool はアニメーションやモデルデータなど、ファイルを展開する処理では必ずといっていいほど使われている。なお、バイナリファイルの場合には new を 1 回にするために、コンストラクタの defaultBlockSize に必要サイズを指定する。最初の 1 回で全量確保するため、それ以上には new は走らない。ただし、アラインメント合わせで無駄になるメモリ量も織り込んで計算する必要がある。

4.12 バイナリデータ互換

機械が複数ある時に、絵や音のデータを共通にするかそれぞれに最適化したものを作るかは意見が分かれるところだろう。

性能を最優先するなら、最適化したものを作るのがいいに決まっている。テクスチャの画素の並び、インデクスパツファの並び、頂点パツファのフォーマット、1 バッチあたりの頂点数、などなど、もともとのデータが同じで同じ絵を出すとしても変えた方がいいことはいくらでもある。おそらく普通はこうやっているはずだ。

しかし、SegaLib では敢えて別の方法を取った。基本的にバイナリフォーマットは一つしか用意せず、全機種で読めるようにしている。Wii 版は根本的に素材が違うので別のファイルにはなっているが、それでも PC や 360 で読み込んで表示することができる。

これは、性能よりも開発の手間を優先したからだ。

PowerSmash4 では、開発は 8 割方 PC 上で行われた。したがって、PC 上で読めているデータをそのまま他の機種でも読めることは開発効率の上で有利である。また、4 機種バラバラにデータを出すとすれば、データは 4 倍になる。サーバの消費容量にしても、データをローカルに持ってくる手間にしても、またデータをそれぞれの機種用に変換する手間にしても、すべてが 4 倍になる。これは正直避けたい。自動化できるにしても所要時間は増すし、管理の手間が全く増えないというわけにも行かないのである。

おそらく、どの機種でも読める形式と、機種専用の両方を用意して、ある程度開発が進んできたら機種専用版も用意し始める、というやり方がバランスの上では良いのだろう。しかし、複数機種向けに同時に作るのは今回が初めてであり、データ管理上の問題まで背負い込むのは得策とはいえない。専用バイナリを用意することで 10% 性能が上がるとしても、たかが 10% のために開発効率を犠牲にして、より価値のある作業をする時間を奪われるのは耐え難い。そして、それでできる性能向上など、たかだか数 % であり、10% を超えるとは考えにくかった。

また、やろうと思えばロードしてからデータを変換する手もあるわけで、ファイルの段階で最適化されている必要は必ずしもない。これはずいぶんと昔の話になるが、パーチャロンマーズというゲームでは汎用的なファイルフォーマットで読み込んだ後、PS2 の GPU に特化した形式に変換する処理をしていた。当時ですらファイル IO の遅さの前では変換にかかる時間などさして問題にはならなかったわけで、今時であればマルチスレッド化もできるのでゲームを止めずにやることもできる。そこで、まずは「共通のままどこまで行けるのか」という方向に進めたわけである。

4.12.1 テクスチャの問題

機種別の最適化を行う上で一番大きいのはテクスチャだろう。これにはテクセルフォーマットと、テクセルの並びの二つがある。

テクセルフォーマット

ARGB8 や DXT1 などのフォーマットは機械によって対応しているものが異なる。もっとも性能を出せるフォーマットに変換してやるのが筋だろう。

しかし、もともと ARGB8 だったものを勝手に DXT5 にして良いかどうかは微妙な問題だ。実機で出してみるまで絵の劣化がわからないというのはこだわりのあるアーティストにとっては許せない問題になる。そういうわけで、PowerSmash4 では劣化を伴うテクスチャ変換はアーティストが自分でやり、プログラマが勝手にやることはない、という原則を設けている。ただ、ARGB8 とは言いつつ色が 256 色しかなかったり、白黒だったりするケースが多くあるため、SegaLib は絵素材のバイナリファイルを生成する際にテクスチャの中身を調べて、劣化がない範囲で容量を減らせるなら減らす工夫はしている。たとえば、白黒の ARGB ファイルは白黒に変換する。色数が少ないなら適切にパレット化する。すべての画素が不透明の DXT5 は DXT1 にする、といった具合だ。この手のミスはかなりの数存在するため、自動でこれらの変換を行うことには効果がある。

また、読み込むマシンが対応していないフォーマットで入っていた場合は、読み込める形式に変換しながらロードする。例えば DXT5 やパレットテクスチャが読めないなら ARGB8 に変換する。変換は本来無駄な処理だが、ARGB8 のまま読むよりもファイル IO にかかる時間が減ることもあり、必ずしも無駄とは言えない。全体的に SegaLib は容量の削減を速度よりも重視している。

このようなしくみであるため、やろうと思えば機種ごとに最適なフォーマットにしてもかわらない。読めないフォーマットであっても変換して読めるからだ。しかし、実際問題そこまでせねばならないケースはなかった。

テクセルの並び

GPU によっては、画素を特殊な順番で並べないと読めなかったり、性能が出なかったりする。左上から右へ行き、右端まで行ったら一段下がって左端からまた始める、という素直な順番のまま GPU に渡して良いことは少ない。

したがってどこかで変換せねばならないわけだが、この変換にはもちろん時間がかかる。バイナリファイルを生成する時に各機種向けに行っておくのがロード時間短縮の上では最良である。これは間違いない。しかし、これをやってしまうと各機種向けに別のバイナリファイルを

吐き出すことになる。先ほどのテクセルフォーマットのように、違う並びだったら変換するようにしておけば、何でも読めるし、事実そうしているのだが、だからといって機種別バイナリを作るのが面倒であることはすでに述べた。

そういうわけで、SegaLib では基本的には普通の左上から右へ進む方式で持っていて、それをロード時に変換している。2048x2048 ともなると変換にかなりの時間がかかるのだが、高速化にそれなりな手間をかけ問題を解決した。GPU がそのための機能を持っているならそれを使ったし、変換をシェーダで書いた機械もある。何も無い場合にはスレッドプールを使って並列化して対処した。結果、機種別バイナリなしでも行けるな、と思う程度の時間にはなった。実際ロード時間のうち変換が占める時間など微々たるものである。これでもなお問題になるようなら、非同期化によって目立たなくする工夫を入れたり、あるいは本当に機種専用バイナリを作ることを検討するだろう。

もちろん機種専用バイナリを作ったとしても、他の機種で読める状況は維持していくつもりである。実際、すでに某機種だけはテクスチャを専用フォーマットで持っているが、これはPCなどの他の機種でも変換して読めるようにしておいた。そうでなければデバグに支障をきたすからである。

4.12.2 頂点データ

結論から言えば、頂点データに関してはいかなる機種別最適化も SegaLib では行っていない。テクスチャと違って画一的な処理が困難であり、さしあたり PowerSmash4 においてはそれほど必要とはされなかった。

複数機種開発においては、通常頂点数は抑えがちになる。ピクセル単位の処理はシェーダと解像度でいくらかでも調整が効くが、頂点はそうは行かない。頂点データを豪華にしすぎると後の修正が非常に厄介である。しかも、プロジェクト開始時にはそれぞれの機械でどの程度の頂点を扱えるかは全くわからなかった。それに、前作の経験もあって、質を落とさずに頂点を減らすこともある程度できたし、頂点数をギリギリまで増やすことにあまり意味はなかった。

結果、頂点キャッシュのヒット率を気にするような状況にはならなかったわけである。そもそも、その手のことを全く気にしていないわけではなく、可能な限り容量を減らしてメモリ負荷を削ると同時に、頂点バッファやインデクスバッファの並び替えは一応は行っている。まったくやらないのに比べればはるかに性能は上がるだろうし、並び替えの際に想定した頂点キャッシュ容量と異なるマシンでそのデータを使ったからといってそれほど遅くなるわけではない。また、スキニング処理を GPU でやるのが不利になる場合には、CPU 側で並列化したスキニング処理を通してから GPU に投げることで入力データを削っている。これだけやっていたら、機種専用の最適化を行う動機はあまりない。

4.13 入力デバイスの問題

ゲーム機間の差異が大きいのはグラフィックスだけではない。むしろ入力デバイスの方がずっと根本的なところで違っている。

しかし、プログラミング上のインターフェイスということで考えるのであれば、ある程度までの統一は不可能ではない。

SegaLib において、入力デバイスは、キーボード、マウス、ジョイスティックの 3 種類である。マウスがない場合もジョイスティックの入力から適当にマウスをエミュレートするため、デバッグ用の UI などと同じ書き方で作ることができる。

ジョイスティックはボタンとアナログから成る。ボタンがいくつあるか、アナログがいくつあるかは機械によるので、数を取得できるし、それぞれの何番が何なのかは機種別の enum に書かれている。加速度センサーやジャイロにしても、結局は数字がいくつか帰ってくるわけだから、アナログスティックの一種としてアナログ配列に入れておくことでインターフェイスを統一させた。単に使いにくいだけで、機種別に関数を用意した方が良いようにも思われるが、それはアプリケーション側でやってもらうこととした。このあたりはあまりにデバイスが多様なので、いちいち最適な設計を考えている余裕がなかったというもある。いっそ入力取得の機能を SegaLib 内に持たない方がいいのではないかとすら考えたが、そうするとあまりに不便であるため、このような形となった。

また、機種によってはモーションセンサーやポインター、振動の有効無効を切り替えることもできるので、そのような関数も持たせてある。機能がなければ単にスルーされる。タッチパネルはポインターの一種として扱い、ポインターの個数を取得して、座標を必要な個数だけ受け取る。

さて、一応はこれでどうにかなっていたのだが、開発途中で彗星のように現れた入力デバイスが二つある。

結論から言えば、これらはアプリケーション側で直接 SDK に触ってもらうことになった。やろうと思えばこちらで扱うこともできたはずだが、開発終盤であったこともあってこちらで対応する余裕が取れなかったのである。今後 SegaLib を使うアプリケーションが現れ、そういった機能を必要とすれば、ここを整備することもあるだろう。

なお、入力周りの実装で最も面倒だったのは、コントローラの状態遷移である。新しく現れたり、切断されたり、電池が切れたり、アナログが無効化されたり、OS のダイアログが出てきて入力を横取りされたり、無線通信状況が悪くて遅延したりと、いろいろなことが起こる。ファイル IO と同じようにこれも IO であり、IO といえばエラー処理と場合分けと相場が決まっている。テストの段になってもここに関するバグはかなりの数検出され、最後まで面倒な思いをした。ハードウェアメーカーの SDK 側でできるだけ処理してくれるとうれしいのだ

が、なかなかそうも行かないようである。

第5章

まとめ

ずいぶん長くなったので、一番訴えたいことを列挙しておく。

完璧はない

現実の状況で、時間と人材の制約の中で作らなければならない。また、それを使って何を作るのか、誰が使うのか、どう使うのか、ということは全て設計に織り込まれる必要がある。したがって、完璧な設計も汎用の設計もありえない。ライブラリは、今使えなくてはならず、使うことが使わないことよりも有利でなければならない。

なお、これを「設計は適当でいい」という意味であると取ってもらってもかまわない。事実 SegaLib の設計はプログラミングの観点から言えば適当であった。しかし、「設計が多少適当になっても良い」という設計をしていた、ということには留意していただきたい。クラスのインターフェイス設計だけが設計ではないということである。

永遠ではない

ライブラリは今の状況に応じて設計され作られる。したがって、現実に関わなくなる時が必ず来る。設計によってはいくらか寿命をのばすこともできるが、それを最優先にしてはならない。また、その時が来たならば、延命をはかるよりも、どうやって終わらせるかを考えた方がいい。

肥大化に備える

ライブラリは肥大化する運命にある。これに対抗して寿命を伸ばすには、積極的に機能を捨てていかななくてはいけない。追加しようとする各機能について、前もって「どういう状況になれば捨てられるか」を考えておく必要がある。また、捨てることが当たり前であるような状況を作る必要がある。そして、この「状況」はかなり広い意味を持つ。ルールであることもあるが、既成事実の積み重ねであったり、あるいは人間関係であったりする。

目的を忘れない

ライブラリの究極の目的は、良いゲームを作る助けになることである。大きく抽象的な目的から順に小さく具体的な目的に落としこんでいく。その過程で上位の大きな目的に反してはいけない。たとえば、優れた技術であってもそれが良いゲームを作る助けにならず保守性を悪化させるのであれば、入れないことを選ばなくてはならない。

目標をバランスする

ライブラリの目標は複数必要だ。性能やコードのエlegantさに囚われがちだが、保守性、ビルド時間、学びやすさ、デバグの容易さ、なども全て品質であり、おざなりにしてはいけない。

つまるところ、ライブラリは道具にすぎず、作ることに意味があるか、というところから問い直さなくてはならない。良い道具とは、最少のコストと最少の複雑さで目的を果たせる道具だ。そうやって割り切った結果、どうにか一人で把握できる範囲で PowerSmash4 用のライブラリを作ることができた。

案外、どうにかなるものである。皆さんもそう思っていたらうれしい。

5.1 SegaLib のこれから

ともかくも SegaLib を使用した初めてのアプリケーションは発売された。SegaLib は子供と言える段階を終えたと言っていいだろう。この後でなお使ってくれる人がいるのであれば、成年期とも言うべき段階に入ることになる。SegaLib の成年期とは、どのようなものであろうか？

まず間違いなく言えるのは、分業しなくてはダメだということである。私は今のところ SegaLib の全てのコードに責任を持っている。逆に言えば、私以外に SegaLib の中身を知っている人間はいない。これは非常に不健全な状況であり、一刻も早く是正する必要がある。ずっとライブラリ仕事をしていれば私は間違いなく腐るであろうし、一人でやっていると独善に陥った時に修正する術がなくなり、また、私に何かがあった時に問題が起こる。しかし、SegaLib 全体を誰か一人の後継者に引き継ぐのは不可能だ。これはスキルの問題ではない。一から書く仕事はまだ面白いが、すでにあるものを受け継ぐ仕事を面白いと思える人はあまりいない。分業によって一人あたりの負荷を下げれば、それぞれの人にとってはそれ以外の仕事もできるので良いが、一人で受け継げばそれだけが唯一の仕事になる。耐えられまい。

分業は、機能別あるいは機種別が考えられる。両方を組み合わせることもあるだろう。アニメーション、システムコール抽象化、入力デバイス、といった機能別に責任者を分けた上で、グラフィクスについては機種別にするのは現実的な選択肢である。ただし、グラフィクスの機種別分業は、複数機種をどう抽象化するか、という全体的な視点を損なう可能性が高く、人材さえ得られるなら一人でやった方が良い。上に描画エンジン部をかぶせるのであれば、それは

また別の人間でもいいだろう。

なお、現状実際の行動は何も起こしていない。しいて言えば、この講演が引き継ぎのための最初の布石である。もっとも、このライブラリを使いたいと言ってくれる人が現れないのであれば、このような悩みは無用のものになる。部分部分のコードは他に持って行って組み込むこともできるので、完全に無駄になるわけではない。このまま SegaLib が名前負けした状態で終わるのであれば、それもまた運命であろう。

5.2 スペシャルサンクス

今回の講演の内容を整理、構成するにあたって、当初私には軸がなかった。起こったことを時間軸に沿って並べているうちに何かが見えてくるだろうと楽天的に考えていた。しかし、7月を過ぎても全体像は見えてこない。相当に焦った。起こった事や、設計や実装についていくら並び立てても、単なる技術発表以上のものにならず、全く面白くないのである。

そんな中、たまたま出会った一冊の本が私に確固たる軸を与えてくれた。昨今「もしドラ」で有名になった P.F. ドラッカーの書籍である。たまたまその「もしドラ」のアニメを見たのだが、15分と見ないうちに「こんなアニメ見てる場合じゃない!」と思ってドラッカーの書籍を図書館に借りに行き、ほぼ全部の書籍を食い入るように読んだ。

今回の発表はドラッカー、そしてその翻訳のノリの影響を濃厚に受けている。

この発表において、あたかも SegaLib が最初から計画的に目的、目標を立てて作っていったかのように話が展開しているが、もちろんウソである。後知恵で歴史を捏造していると言っていいレベルにウソである。もちろん、保守性を重視するとか、人材と時間が足りないとか、アプリケーションがバグを出しにくい作りにしようとか、そういったことをバラバラには考えていたし、言葉にはならないまでも大まかな方向性というものを考えてはいた。目標を複数立ててそれらをバランスしようという思想も最初からあった。実際、上司にプレゼンするために書いた文書を今見てみれば、それらのことが一応の論理性を持って書かれてはいる。しかし、ライブラリがゲームを作るための道具にすぎない、という大きな視点は実のところここ2ヶ月の間に得たものであり、バラバラに考えていたことが一つに結びついたのはそれによってである。したがってこの発表は根本的な所にウソがあると言って良い。だが、敢えてそのような話の流れにした。

そういうわけで、P.F. ドラッカーと、その著作をあかも素敵な日本語にしてくれた翻訳者には深く感謝している。そして、みなさんにもおすすめしておきたい。「経営者の条件」がもっとも薄くて読みやすい。何年かプロとして仕事をした人であれば、50ページも読めばもう止まらなくなっていることだろう。過去にしてきた仕事のことが走馬灯のように思い出されるに違いない。なお、書名には「経営者」とあるが、とりわけゲーム会社のように知的な労働が核になる世界ではほぼ全員に当てはまる。中でもプログラマは最も良く当てはまるように思え

る。強くお勧めしておきたい。