Android向けミニゲーム制作ライブラリ

Amanatsu  を作ってみた

Hiroki

これは Android 向けの OpenGL を用いたゲーム制作ライブラリ Amanatsu の開発記事です。

1 Mikanっぽいライブラリ

Android でゲーム開発をしようと思った時に立ちふさがるのは、端末ごとの性能差や GC[1]描画方法だったりします。X680x0 同好会でも何度か簡易講座をやってみたものの、あまり普及しませんでした。そもそも私自身も毎度毎度 FPS[2]制御や描画管理等を行うのは面倒なのでやりたくありません。やはり FPS 制御をざっくりとやってくれ、準備フェーズなどが用意されたライブラリが欲しいわけです。そんな気軽に使えるのがない？ならば作りましょう。それが開発者というものです。

そんなわけで、今回は Amanatsu[3]と名付けた Android 用のゲーム制作ライブラリを作ることとしました。このライブラリは私と何人かの協力者で作っている PC 向けゲーム制作ライブラリの Mikan[4]系列の新ライブラリという位置付けで、基本的な設計方針等を取り入れていきます。

しかし、ライブラリを作ると言っても、最近 Android 版もフリーになった Unity や 2D ゲームを作成する Cocos2d-x などの高機能なゲーム制作ライブラリがたくさん出てきています（正確には、C++を用いたクロスプラットフォーム向けの開発環境）。ただゲームが作りたいなら、そこら辺の高機能で流行ってるものを扱えるようにした方がいいので、独自ライブラリを作る前にコンセプトを明確にする必要があります。

そこで初めにコンセプトを設定しました。今回のコンセプトは単純明快、たった１つです。

- Java のみで使える。

Java のみで使えるとは、JAR[5]単体で読み込ませるだけで使えるようにするということです。なぜこれをコンセプトにしたかといえば、最近の Android の開発環境事情が関係してきます。スマートフォンは高機能・高性能なため、最近ではお絵かきや作曲のみならず、アプリ制作や DAW による音声加工も可能になってい

[1]不要になったメモリを開放する処理。

[2]1 秒間に何回画面を更新するかの単位。

[3]Mikan 系列のライブラリには柑橘系の果物の名前を与えることとしました。今回は Android の頭文字である A を含む甘夏を採用しました。

[4]C++向けの DirectX を用いた PC 用ゲーム制作ライブラリ。詳しくは 2011 年の 68 通信 (http://www.x68uec.org/other/press/2011/) や MikanWiki(http://mikan.azulite.net/) を参照してください。

[5]Java のアーカイブで Java の class ファイルや画像などのリソースを ZIP 圧縮したもの。

ます。つまり、Android 端末のみでゲーム開発するのも夢ではありません。キーボードさえあれば実用的範囲まで来ていると言っても過言ではありません。そこで AIDE 等の Android 上で Android アプリを開発するような開発環境でも使えるよう、Java 単体で使えるライブラリが必要となります（余談ではあるが、libs フォルダ以下に JAR ファイルをコピーするだけでそのライブラリを使えるため、とても手軽）。

また、Java のみであることと Android 上で動かすことが前提となると、処理スペックの問題が立ちふさがります。正直な話、低レベルな部分から直書きして、ちゃんと GC 等の挙動を把握したプログラミングができれば何とかなるのかもしれません。しかし、大抵の人はそこまで深く言語を理解するわけでもないですし、かと言って何も考えず作れば多くのゲームは処理が重くなってしまいます。そこで今回はハイスペックなゲームも作れるなどという目標は持たず、ミニゲームにターゲットを絞ることになりました。

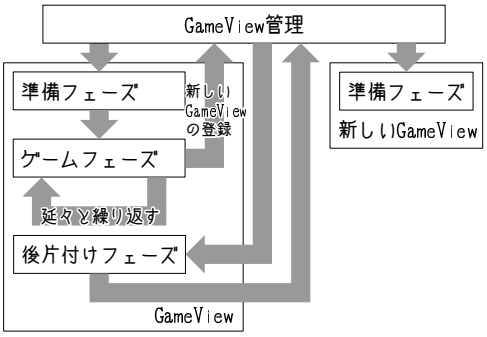
そんなわけで、Android 向けミニゲーム制作ライブラリ Amanatsu を作ったので、その制作の様子というか設計及び苦労話をまとめていきたいと思います。

2　ライフサイクル

ライブラリのライフサイクルは重要です。大元となっている Mikan ライブラリは準備フェーズが 2 つ、延々と 1 秒間に 60 回実行されるゲームフェーズ、そして後片付けのフェーズの合計 4 つのフェーズがあります。2 つの準備フェーズを実行し、ゲームの終了処理が呼ばれるまで延々と同じ処理を繰り返し続け、最後には後片付けを行ってプログラムを終了するというのが、Mikan ライブラリのライフサイクルとなります。

今回は上のフェーズを準備、ゲーム、片付けの 3 つとします。また Android のアプリにも独特なライフサイクルがありますが、それを隠蔽化するのはとても面倒なので、それとは別に管理します。さらに、今回の動作環境は Java というオブジェクト指向言語なので、クラスを扱えるのは必須条件となります。それもあり、ライフサイクルの処理をまとめた `GameView` クラスを用意し、それを継承したものを登録することで Amanatsu のゲームサイクルを管理することとしました。これによりシーン遷移も `GameView` クラスを継承したシーンごとのクラスを Amanatsu に登録して切り替えられるようになり、ゲームをシーンごとにクラスで作成しやすいようになりました。

ざっくりと図示すると以下のようになります。



ただし、Android のライフサイクルが若干特殊なので、方向固定の設定を行わ

ないと端末の方向を変えただけでゲームが再起動してしまったり、ゲームを別のアプリに切り替えると終了してしまうこともあります。そのため、このライフサイクルは絶対に守られる類のものではなく、ゲームを作りやすくするものと割りきって扱う必要があります。

3 描画

描画には OpenGL を採用しました。Android には Canvas、SurfaceView、OpenGL の 3 つの描画方法が提供されています。Canvas は簡単ですが描画が重く、SurfaceView と OpenGL は描画の処理が面倒です。SurfaceView は描画能力は Canvas と同程度のようなので、同じく面倒ならより描画速度が早く 3D も扱える OpenGL[6]を採用することとしました（DirectX で得た知識という資産を扱えるメリットもある）。ゲームライブラリの作成においてこの描画は最も重要な部分となります。使い方は Mikan ライブラリと似たような使い方になるようにしつつ、Mikan ライブラリの引数の不具合を修正することとしました。また、Java や OpenGL 特有の問題も多々発生したので、いくつか重要な部分について書いていきたいと思います。

3.1 引数

実際の描画を行う引数は画像の番号や読み取り開始座標、大きさ、描画先座標や描画情報を扱う必要があります。かなり多くの値を扱うことになりますが、画像の一部分を切り取って画面の任意の場所に貼り付けるにはこの程度の情報が必要となります。Mikan ライブラリでは読み込みと描画の引数関係がぐちゃぐちゃだったため、今回は読み込み情報を先、描画情報を後になるようきっちりと設計しなおしました。読み込み情報が先の理由は、読み込みは開始座標と大きさという要素が増えたり減ったりはしないからです。これにより多少は引数の与え方に秩序が持たされるようになりました。また、Mikan 系列のライブラリの特徴である、引数のパラメーターを増やすとより詳細に描画を指定できる特徴をより理解しやすくなりました[7]。

3.2 画像の読み込み

グラフィックボード (以下グラボ) を使って高速描画するためには、グラボのメモリに画像のデータを転送する必要があります。そのデータの管理方法として、正方形かつ 2^n の画像サイズでなければならないというルールがありました。最近のグラフィックボードで DirectX を使用する場合はそのような制限はなくなりましたが、Android の OpenGL ではその条件が必須となります。

このようなルールがあるため、ユーザーにその制限を課したいところではありますが、それでは知らない人がいつまでも正常に描画できないことが想定され

[6] グラボを使って 3D や 2D 描画を行う API。Android 等の携帯機器に搭載する場合用の組み込み向けの OpenGL ES が利用されている。

[7] 実際の利用者の声

るので、とても不親切です。今回は無理矢理にでも描画できるように、必要があれば画像の大きさを Amanatsu 側で修正することにしました。今回はミニゲーム用のライブラリで画像の一部分を切り取って使う利用方法ということで、拡大縮小は行わず、画像サイズから正方形かつ 2^n の画像サイズに満たない部分を透明な色で埋め尽くすことで画像サイズを適正な大きさにします。

また、Android での画像などのリソースに関しては、2 つの保存方法があります。res フォルダ等の決められた場所にファイルを入れる方法と、assets フォルダを作りその中に入れる方法です。前者は画像ファイルが端末のサイズによって勝手にリサイズされてしまうので、assets フォルダに入れる方法を推奨することとしました（res 内の画像も扱えるようにはしている）。

ちなみに捕捉ですが、Android 搭載端末はハードウェアの個体差が非常に激しいです。もちろん OpenGL で扱える画像サイズの限界も様々です。そうなるとどの程度の画像サイズであれば扱えるのかが非常に重要となりますが、Android の OpenGL では 1024 のサイズまでなら作れるようにすることと明記してあるため、基本的には 1024 までのサイズであれば作れると考えてよいでしょう。

4　入力と音声

ゲームライブラリでは入力や音声も重要な要素です。描画ほどではないですがこちらもいろいろと面倒な問題が出てきます。

4.1　入力

Android の入力はタッチとその他センサーの 2 種類に分かれます。Amanatsu ではその他センサーに関しては傾きを検知する機能だけ管理しています。傾きは Android の API で Roll、Pitch などの名前で簡単に取得できる値でしたが、Android の端末と API がしっかりと連携が取れていない関係上、このやり方では端末によって得られる値が全く異なるという事態[8]になりました。そこで、少し面倒ではありますが磁気とジャイロセンサーから端末の傾き具合を検出する方法[9]を採用しました。これにより端末間の差異を考えなくてもよくなります。

タッチ入力についてはタッチしているフレーム数を返す管理機構を作成しました。これにより、より長くタッチしているなどの情報が入手可能です。一方マルチタッチを考慮した場合、指には ID が割り振られるため、それを元にハッシュで管理することでユーザーがマルチタッチでも個別の指を認識できるようにしました。その際、後述するメモリの問題もあるため、あまり大きくないハッシュサイズにしつつも、ハッシュテーブルの大きさが足りなくなって再確保するようなことがないサイズにする必要があります。こちらは指が 10 本までと仮定し、ハッシュテーブルの領域再確保の目安の数を調べた上で適切に設定を行いました（資料を

[8] 多分開発会社の認識の違い。
[9] このやり方を推奨。少し面倒だが処理のサンプルはたくさん転がっている。

紛失してしまったが、確か現在のテーブルサイズの 2/3 を超えると領域の再確保が行われる。10 本を上限として超えない事を考えると 16 が安全最小値となるので、Amanatsu では 16 に設定している）。

ついでではありますが、Android ではキーボードやゲームパッドの入力にも対応しているので一応対応させておきました。キーボードの上下左右キーとゲームパッドの上下左右ボタンが同じ割り当て番号だったりと少しおもしろい関係になっていたことと、Windows の仮想キー割り当てのように QWERTY キーの左上から順番ではなく、（プログラミングしやすいようになのか）A〜Z が連番になっていたことを捕捉しておきます。

4.2 音

音は 3 つの方法で再生が可能です。BGM のように長い曲を再生する MediaPlayer、効果音のように短い音を再生する SoundPool、そしてバイトデータを再生する AudioTrack です。初めの 2 つは AudioTrack を利用しているとのことなので、大元は AudioTrack のようです。しかし、AudioTrack は自分で音声ファイルの解析やループ制御等を行わなければならないので、とても面倒です。MediaPlayer は再生に遅延が生じたり、SoundPool は長い音を再生できないなどデメリットは有りますが、この 2 つを採用することとしました。特に SoundPool は端末ごとの再生可能秒数が異なり 1 秒程度でないと安心できませんが、AudioTrack での実装は時間もあまりかけたくなかったのでやめました。後日音について何とかしたいと感じたら、本腰を入れて AudioTrack を用いた再生機構を作ろうと考えています。

4.3 GC 対策

Android でゲーム制作といえば、最大の敵は GC です。GC はメモリ管理機構のことで、メモリが圧迫されてきたら不要になったメモリ領域を使えるようにするありがたい機能です。最近の言語にはデフォルトで採用されている場合がほとんどです。これの問題点は、この GC の作業が少し重いのと、アプリの実行を止めてしまうことです。ユーザーが書く部分に関してはどうしようもありませんが、ライブラリ部分は GC を積極的に起こさない工夫をする必要があります（少し話は変わるが、C#では GC を任意タイミングで実行可能。そのような機構がある場合は、シーン遷移など少し止まっても問題ない場所で GC を積極的に行うという選択肢もある）。

では、GC を起こさないためには何をすればよいでしょうか？　答えは簡単で、変数の使い回しを行うことです。入力のところでハッシュテーブルのサイズを最適に……などと書いていましたが、これも GC 対策の一環です。メモリの再確保＝不要になるメモリ領域があるということですし、不必要に大きなテーブルを確保するとメモリが足りなくなって GC が発生してしまうので、極力それを避けるよう努力します。簡単な部分で言えば、ループ変数を全てメソッド内ではなくクラス内に定義します。こうすることでインスタンスを削除しない限りはメモリを不要に確保しなくなります。特に OpenGL で

は float 配列を 3D の描画に使いやすいように FloatBuffer というものに変換して扱うのですが、それも作った後は再利用します。具体的には値は `put()` メソッドで書き込み後は必ず `position()` メソッドでバッファーの位置を 0 にします。また float 配列も予め必要個数を確保しておき、その配列にデータを書き込んでから FloatBuffer に `put()` メソッドで書き込みます。

こんな対策で効果あるのかと疑問に思う方もいるかもしれませんが、FloatBuffer に対してこの改善を行うだけで処理落ちが緩和されました。例えば 1 枚の画像を貼り付けたとき、座標、頂点色、UV 座標で FloatBuffer 4 頂点分使うので、少なくとも $3 \times 4 \times 4$ バイトは消費することとなります。Amanatsu のデフォルト FPS が 30 なので、1 秒間に約 1KB は消費することとなります。実際にはもっといろんな場所でメモリを確保しまくっていると考えられるので、その数倍にはなるでしょう。携帯端末はそんなにメモリ容量があるわけではないのでそれだけのメモリ領域を毎秒確保していると、あっという間に GC が実行されてしまいます。アプリの実装言語である Java は C 言語などと比べてリッチではありますが、動かす環境が PC などと比べると貧弱ですので、メモリ管理は重要な要素となります。

捕捉ですが、Android のバージョン 2.3 以降では、平行 GC という GC が採用されています。この GC を採用しているバージョンでは GC によるアプリ停止がかなり緩和されています。しかし、ゲームでは ms 単位の処理の遅れが致命的であり、平行 GC を採用したとしても対策なしでは処理落ちが多々確認されています。GC 対策は頑張りましょう。また、NativeActivity などを用いて Java の世界を抜けだして C++で実装すれば、GC の影響下から逃げることも出来ます。恐らく他の C++で Android ゲームを作れるライブラリは、これを利用していると思われます。

5 その他

実装の大体の苦労部分は終了しました。後はいくつか補足的な内容について書いていきます。

5.1 描画捕捉

Windows + DirectX と Android + OpenGL では画面の座標の設定方法が異なります。具体的には Windows + DirectX では左上原点 Y 軸下方向ですが、Android + OpenGL は左下原点で Y 軸上方向です。ここは Mikan ライブラリが左上原点なのでそちらに合わせるようスクリーンの設定を行いました。これに加え、描画の向きも標準では DirectX と OpenGL の向きが異なるため、ある種の資産再利用のために逆向き（Mikan ライブラリと同じ方向）にしました。

また、描画モードについては、加算や乗算は良いのですが、減算などが OpenGL のバージョン依存のようなので、減算は今のところ用意しないこととなりました。Android のバージョンごとのシェアからそこら辺の機能拡充について考える予定です。

次に、このライブラリは画像の描画をある種のメインとしているため、基本図形の描画は若干重くなる可能性があります。というのも、基本図形と画像の描画は根本では同じ板ポリゴンの描画で実装されています。しかし、内部的には画像を貼り付けるモードとそうでないモードを毎回切り替える必要があります。OpenGL的にはそれらをちゃんと設定する方がよさそうなのですが、今回は基本画像描画で設定を保持し、基本描画時のみその設定を無効化して描画し、再度有効化するため、若干処理が重くなるのではないかと考えられます。また、これらの事情によりオープンソースで公開しているのですが、一部分を切り取って使っても正常に使えない可能性があったりします。今後の課題として、ここら辺の実装についてはもっと調査して適切な状態にしていくべきでしょう。

5.2 ライブラリについて

ライブラリは初めに書いたようにJARファイル単体のみで稼働するようにしました。AIDEといったAndroid上でAndroidアプリを作るアプリでもlibsフォルダにAmanatsuのJARファイルを入れ、適切にimportしたプロジェクトを作ったところ、無事稼働しました。これで当初の目的は達成しました。

また、起動時にデフォルトではAmanatsuのロゴを表示しています。これは無効化も可能なので単純なお遊びでもあるのですが、JARファイル内に入れておいた画像を描画するテストでもあります。

さらに、このライブラリは先ほど書いたようにオープンソースです。GitHubにて公開しており、誰でも改変及び利用が可能です（利用だけならJARファイルをダウンロードするだけで良い）。

6 最後に

大体このような形で開発が進みました。ざっくりと１週間程度で基板ができ、次の１週間で最低限使えるものに仕上げ、２０１３年度の春合宿でサンプルゲームを作れる規模になりました。全体では約１ヶ月ほどかかりました。OpenGL直叩きは初めてでしたが、DirectXよりは準備も少なく、またDirectXで培った基礎知識と楽なライブラリの揃ったJavaだからこそ、この期間で使い物になるものができたかなという感想です。特に画像の大きさ補正やハッシュなどでは大いに楽が出来ました。やはりオブジェクト指向が前提の設計、豊富なライブラリや機能がデフォルトで提供されていると開発がとても楽で良いです。

一方メモリ管理やライフサイクルには苦しめられました。Javaにはデストラクタが無いため終了時に処理を行うには一工夫必要ですし、GCは専用の項目があるくらいには苦労しました。それでもアプリ終了はAndroid任せなので、後片付けフェーズは正常に実行されないことが多々あります。細かなチューニングが必要な場合には言語やGC等の仕組みを理解していなければいけませんし、場合によってはいろんな部分を自分で実装する必要もあるので、そこら辺はGC搭載言語の悪い部分でしょう。

今後についてですが、Mikan 系列のライブラリは私という需要がはっきりとありますので、いろいろな方面に勢力を拡大させつつあります。Amanatsu の後は WebGL という OpenGL の Web 版を JavaScript で使えるようにしたものがあるので、それらを使ったライブラリも開発中です。本当は JavaScript には明確な型情報がなく、クラスなども非常に面倒な実装方法になるので私は大規模開発に手を出したくありませんでした。しかし、TypeScript[10] と呼ばれる JavaScript の上位言語で実装し、JavaScript に変換する方法を採用することで、楽に開発をすすめることに成功しました（変数名や構造もほぼ残るので分かりやすい）。残念ながら開発し始めてすぐ WebGL の動く端末がないことに気がついたので作業凍結中ではありますが、FPS 制御及びライブラリの基盤はできました。WebGL が動く端末を入手次第、続きの実装を TypeScript で行っていく予定です。さらに、最近 PSVita を入手し、ちょうどライセンスキーが 1 年無料となるキャンペーン中だったこともあり、そちらでも似たようなライブラリの作成を行っています（忙しくて出来はまだまだですが）。

これから先、個人でも携帯端末やゲーム機での開発がどんどん可能になっていくでしょう。数年前には考えられなかったことです。Web もリッチになっているので、いろいろなプラットフォームでゲームが作られることでしょう。私は今年で X680x0 同好会を卒業しますが、ここで学んだことを生かし、少しでもいろいろな環境で、ゲーム制作を楽しんでいきたいと考えています。

7　リンクとか

Amanatsu - GitHub　- https://github.com/HirokiMiyaoka/Amanatsu

Amanatsu 本体のダウンロードやソースファイルなど。利用する場合は Amanatsu.jar をダウンロードして下さい。

68 通信 vol.17（2011 年度版）　- http://www.x68uec.org/other/press/2011/

Mikan ライブラリを作ってみた記事があります。

[10] Microsoft が開発した。VisualStudio 以外でも Emacs や Vim、SublimeText で連携可能。