電気通信大学 MMA の活動情報誌

hyakumangoku 百萬石

Spring 2013

実録!? Genius Bar
hirosun

キーボードマニア
mznh

つくる! WM
clear

掃き出そう
zico

Androidのスレッディング機構について
alstamber

オシロスコープで図形を描こう!
mermao

Twitterのリストを自動生成してみる
hayakawa

MMA

部長挨拶

百萬石を手に取って頂きありがとうございます。MMA は主にプログラミング及びネットワークに関する管理・開発を行なっているサークルです。昨年度は、プログラミングについては、個人で様々なもの[*1]を作っている他、チームを作って開発や[*2]、プログラミングコンテストへの参加[*3]、ネットワークについては、サークル棟や部内のネットワークの管理[*4]やセキュリティに関する勉強及び大会への参加[*5]などを行いました。

部室はサークル会館の 2 階にあります。興味を持っていただけた方は気兼ねなく部室に来てください。

2013 年 4 月
部長　nomeaning

[*1] Window Manager, Twitter Client, Unix Shell などを作っている人がいました
[*2] Kagisys(部室の電子鍵システム) や、WikiPlugin 開発などが主なものです
[*3] 今年の 7 月にロシアで開かれる ICPC の世界大会にも参加します
[*4] MMA はサークル棟ネットワークの管理を委託されています
[*5] SECCON CTF に参加し、全国大会 5 位でした

目次

Android のスレッディング機構について (alstamber) **1**

実録!? Genius Bar (hirosun) **9**

キーボード マニア (mznh) **12**

つくる! WM (clear) **16**

オシロスコープで図形を描こう！ (mernao) **27**

掃き出そう (zico) **34**

Twitter のリストを自動生成してみる (早川　敦士 (hayakawa)) **37**

Android のスレッディング機構について

alstamber

はじめに

Android 開発をする上で避けては通れないのがマルチスレッディングである。外部との通信やデータベース処理などの重い処理をメインスレッドとは別のスレッドで行うことはユーザービリティの向上の上でも不可欠である。そこでこの記事では Android がどのようにスレッドを取り扱っているのかについて述べたいと思う。

1 注意

以降 Android の開発言語としては Java を使用することを前提とする。Scala なんて使ってる変態さんは知りません !! あと Java についてそれなりの知識があることも前提とします。

2 Android における基本的なスレッドの扱い方

Android でスレッドを扱う基本的な手法は Java のそれと同じである。例えば新しいスレッドを開始するには次のようなコードを書けば良い。

```
new Thread(new Runnable(){
  @Override
  public void run(){
    //あーだこーだ
  }
}).start();
```

まずは理屈っぽい説明から。`Thread` というクラスのオブジェクトを生成し、それの `start` というメソッドを実行するという形になっている。この `Thread` クラスは Java においてスレッドを抽象化して扱うためのものである。`Thread` クラスのコンストラクタはそのスレッドで実行する処理を含んだクラスを引数として取る。このクラスは `Runnable` というインターフェースを実装していなければならない。`Runnable` インターフェースは `run` という抽象メソッドを持っているので、`Runnable` インターフェースを実装したクラスは必ず `run` を実装しなければならない。`run` はそのクラスで実行される処理を表すメソッドである。よって上のコードを実行すると新しいスレッドが生成されて、そのスレッドの中で `Runnable` インターフェースで定義されている `run` メソッドの内容が実行される、ということになる。

これじゃわかりにくいかもしれないので、もっと噛み砕いた説明を。イメージとしてはスレッドでやりたいことを書いたクラスを用意して、それを新しく作った `Thread`ってやつに放り込んで `start` すると新しいスレッドで処理ができちゃう、という感じで良い。スレッドでやりたいことを書いたクラスというのはなんでもいいわけじゃなくて、`Runnable` というインターフェースを実装していないといけないことになっている、という感じ。なんで `Runnable` を実装しなければいけないのかというと、`Runnable` を実装すればそのクラスには確実に `run` というメソッドがあることが保証されるから。プログラムを実行するとき、新しいスレッドを作りなさいという指示を見つけたら `run` メソッドの中身をスレッドで実行する内容として採用す

る、という仕組みになっているわけである。

Java に準拠したスレッド処理機構を持っているため、synchronized や wait、notify といったものも使うことができる。synchronized は排他制御と呼ばれる制御を行うためのものである。synchronized は次のように使用する。

```
public void func1() {
  synchronized(this) {
    //あーだこーだ
  }
}

public void func2() {
  synchronized(this) {
    //どーのこーの
  }
}
```

中括弧で囲まれた部分がブロックとなり、synchronized の対象となる。synchronized にはロックオブジェクトというものを指定する。synchronized のブロックが属するスレッドは synchronized ブロックを実行するときに、指定されたロックオブジェクトの占有権を取り合う。そして無事占有権を獲得出来ればスレッドはブロックの中身を実行できる。すでに別のスレッドが占有権を獲得していて占有権を獲得できなかったときは獲得できるまで待つ処理をする。こうすることによってロックオブジェクトに指定されたオブジェクトは常にひとつのスレッドからしかアクセスされないことが保証される。これによって処理の途中に別のスレッドによってデータが破壊されるなどといった悲劇を防ぐことができる。

これに絡んで wait と notify というメソッドが登場する。synchronized ブロック内で wait を使用すると一旦そのスレッドは停止し、別の待っているスレッドに処理を渡す。notify メソッドを使用すると wait メソッドによって停止したメソッドのうちどれか一つが (どれが選ばれるかはわからない) 再開される。wait メソッドによって停止したメソッド全てを再開したいときは notifyAll メソッドが使える。

3 Android の GUI 周りのスレッド処理

先程述べたように Android は Java の手法に基づいてマルチスレッド処理をすることができる。ネットワーク通信や I/O などの重い処理はメインのスレッドではなく別のスレッドで行うべきである。重い処理をメインのスレッドでやるとメインのスレッドで UI の処理ができなくなってしまい、UI が固まってしまったように見えるからである。そもそも Android3.0 以降ではネットワーク通信をメインのスレッドでやると NetworkOnMainThreadException が発生するようになった。つまり Android3.0 以降ではネットワーク通信は「必ず」別にスレッドを作ってやらないといけないことになっている。

一方で UI 周りの処理は逆に「必ず」メインのスレッドでやらなければならない。メインのスレッド以外から UI 周りをいじろうとすると CalledFromWrongThreadException が発生する。Android の UI はシン

グルスレッドモデルになっているのである。

なぜシングルスレッドモデルになっているのか。これは GUI 周りの処理というのは非常に複雑になりがちであるからというのが一例として挙げられる。GUI をマルチスレッドで処理することを考える。GUI の部品は複数のスレッドからアクセスされると壊れてしまう可能性があるので先ほど挙げた synchronized などを使ってロックを掛けて処理する必要がある。さらに GUI においては部品同士に親子関係があり (ボタンとそのボタンが配置されているビューなど) 親から子のやり取り、子から親のやり取り、更にはアプリケーションのコードから GUI のライブラリのコードを実行したり、その逆があったりと煩雑な処理をしなければならない。この時ロックの仕方について注意深く留意しないとお互いにロックの解除を待ちあって永遠に処理が進まないデッドロックがしてしまうなど、バグの発生しやすい環境になってしまう。

そこで思い切って UI の操作はメインのスレッドからしか出来ないことにして、このような問題が発生しないような設計になっているのである。

4 Handler

とは言えネットワークからとってきたデータを UI に反映させるなど、メインスレッド以外から UI を変更したいという需要は存在する。そのために Android には Handler という仕組みが用意されている。Handler は次のように使用する。

```
final Handler handler = new Handler();
new Thread(new Runnable(){
  @Override
  public void run(){
    //あーだこーだ
    handler.post(new Runnable(){
      @Override
      public void run(){
        //どーのこーの
      }
    });
  }
}).start();
```

上のソースコードで「あーだこーだ」と書いた部分は別のスレッドで実行されるが、「どーのこーの」と書いた部分は handler を new したスレッドで実行される。handler をメインスレッドで new すれば、別スレッドからメインスレッドに処理を投げることができる、というわけである。

5 Handler のしくみ

Handler の仕組みを知るために、実際に Android のソースコードを読んでみることとしよう。Handler.java を紐解いてみると post メソッドは次のように定義されている。

```
public final boolean post(Runnable r)
{
  return  sendMessageDelayed(getPostMessage(r), 0);
}
```

sendMessageDelayed メソッドと getPostMessage メソッドが見えるので、続いて sendMessageDelayed メソッドをみてみる。

```
public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
  if (delayMillis < 0) {
    delayMillis = 0;
  }
  return sendMessageAtTime(msg, SystemClock.uptimeMillis() +
      delayMillis);
}
```

```
private final Message getPostMessage(Runnable r) {
  Message m = Message.obtain();
  m.callback = r;
  return m;
}
```

getPostMessage メソッドは受け取った Runnable を実装したインターフェースを基に Message オブジェクトを生成しているようだ。この中で実行されている Message.obtain メソッドは予め用意された Message オブジェクトのプールから使っていないものを取り出してあてがうという処理をしていて、処理の足かせになりがちな無駄な new 文実行の回数が減るように工夫されている。sendMessageDelayed メソッドは内部的には sendMessageAtTime メソッドを実行しているだけのようだ。このメソッドをみてみると次のようになっている。

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
  boolean sent = false;
  MessageQueue queue = mQueue;
  if (queue != null) {
    msg.target = this;
    sent = queue.enqueueMessage(msg, uptimeMillis);
  }
  else {
    RuntimeException e = new RuntimeException(
    this + " sendMessageAtTime() called with no mQueue");
    Log.w("Looper", e.getMessage(), e);
  }
  return sent;
}
```

MessageQueue なるクラスに定義されている enqueueMessage というインスタンスメソッドを実行しているようである。これがどうやら Handler の本質っぽそうだ。この MessageQueue というのはその名の通り Message が整列しているキューである。紙面の都合上掲載しないが enqueueMessage メソッドでは引数に指定された Message を新たにこのキューに追加する処理をしている。この時もうひとつ与えられた引数である uptimeMillis の数値が Message オブジェクトが持つ when というフィールドに渡される。そしてすでにキューに登録されている Messsage の when と、新しい Message の when を比較し、when の小さい順に並ぶように並び替えている。この when は Message をいつ処理するかという時刻を表していて、この時刻に到達した時点で Message はキューから取り出されて処理される。つまり when の小さいものから順番に処理されていくということになる。

6 Looper

ではそのキューから Message を取り出して処理する仕事は誰がやっているのだろうか。それをやっているのが Looper である。Looper には loop というメソッドが存在している。このメソッドは次のようになっている。

```
public static final void loop() {
  Looper me = myLooper();
  MessageQueue queue = me.mQueue;
  while (true) {
    Message msg = queue.next(); // might block
    if (msg != null) {
      if (msg.target == null) {
        return;
      }
      if (me.mLogging!= null) me.mLogging.println(
        ">>>>> Dispatching to " + msg.target + " "
        + msg.callback + ": " + msg.what
        );
            msg.target.dispatchMessage(msg);
            if (me.mLogging!= null) me.mLogging.println(
              "<<<<< Finished to    " + msg.target + " "
                + msg.callback);
            msg.recycle();
        }
  }
}
```

ご覧のとおり loop は無限ループになっていて、その中で MessageQueue の next というメソッドを呼び出している。紙面の都合上、コードは示さないが、この next は Message の when を調べて、キューから取り出しても良いと判断された Message のうち一つを取り出すメソッドである。そしてその Message の target フィールドに対して dispatchMessage というメソッドを実行している。先ほど示した sendMessageAtTime メソッドを見ればわかるが、target フィールドは Message を生成した Handler を示している。そこで Handler クラスに定義されているであろう dispatchMessage メソッドを調べてみることにする。

```
public void dispatchMessage(Message msg) {
  if (msg.callback != null) {
    handleCallback(msg);
  } else {
    if (mCallback != null) {
          if (mCallback.handleMessage(msg)) {
            return;
          }
    }
```

```
    handleMessage(msg);
  }
}
```

Message の callback には getPostMessage メソッドを見れば分かる通り、Handler.post メソッドに渡した Runnable が入っている。そこで handleCallback メソッドが実行されるとわかるのでそれをみてみる。

```
private final void handleCallback(Message message) {
  message.callback.run();
}
```

このように Runnable の run メソッドが実行されていることがわかる。Handler はメインメソッドで生成されたため、このメソッドもメインメソッドで実行される。こうしてめでたく Looper によって Message が取り出された後、run メソッドがメインスレッドで実行されるまでの流れを知ることが出来た。

7 Zygote と ActivityThread

ここまで来て更に気になるのは Looper なるものはどこから湧いてくるのか、ということだろう。これを知る上で Android においてどのようにプロセスが生成されるのかについて知る必要がある。Android のプロセス管理において重要になるのが Zygote というプロセスである。Zygote は全てのアプリケーションプロセスの親プロセスとなるプロセスである。Android には Activity Manager というサービスがあり、このサービスが Zygote とやりとりをしてアプリケーションプロセスの生成が行われる。

まず Android 上でアプリケーションの起動要求があるとその要求を Activity Manager が受け取る。Activity Manager は要求を満たすアプリケーションを調べてプロセス生成の要求を Zygote に出す。Zygote は fork システムコールを実行して新しいプロセスを生成する。新しいプロセスは自動的に ActivityThread というクラスのコードをロードする。ここには main メソッドが定義されていて、Java の規定に従い main メソッドからコードが実行されていく。ActivityThread は Activity Manager とやり取りをしてアプリケーションの起動に必要な処理を行い、起動すべき Activity を決定、その Activity のインスタンスを生成して onCreate メソッドを実行しアプリの実行がスタートする、という仕組みになっている。

この ActivityThread の main メソッドで Looper の準備が行われているというのが真相である。main メソッドでは Looper.prepareMainLooper() というメソッドが実行されている。このメソッドはメインスレッドに対して Looper と MessageQueue を用意するメソッドである。MessageQueue は Looper に属するようになっていて、この Looper は ThreadLocal に保持されている。ThreadLocal はスレッドごとにオブジェクトを保持するための仕組みである。今の Looper.prepareMainLooper() の場合なら、メインスレッドが Looper を保持し、その Looper が MessageQueue を保持するような構造となっている。メインスレッドで生成された Handler は自分のスレッドが持っている Looper に属している MessageQueue に対して Message を登録し、それが Looper によってメインスレッドで実行される、という仕組みになっているのである。

そして Looper の準備ができたあと Looper.loop() が実行され、前述の Message の処理の準備が整うことになる。

以上のような仕組みになっているため、実は自前のスレッドに Looper を用意すれば、それに Handler をつかって処理を投げることができる。Handler はメインスレッドに UI の処理を投げるために使われることが非常に多いが、もっと一般的にスレッド間通信に使用することも可能なのである。

実録!? Genius Bar

hirosun

はじめに

購入して一年もしないうちに私の MacBook Pro 15in の液晶が残念ことになってしまったので Genuius Bar に修理に出しましたというお話。Genius Bar の利用レポートのような何か。

1 液晶の故障、そして Genius Bar へ …

あれは 2 月 8 日、金曜日の夜の出来事でした。MMA の部会を終え、夕食の後帰宅、珈琲を淹れて机に向かい、MacBook を開くと …… その液晶は全体が緑色のノイズに晒されていました。「えっ？ちょっと待って、二時間前はなんともなかったじゃないですか?! 」などと動揺しつつも本当に液晶がダメになっているかどうかを確認するためにスクリーンショットを撮り、別の端末で表示させてみると …… なんともない、スクリーンショットの画像を見る限りは正常であり緑色の奇妙なノイズは見られない。ああ、液晶が逝ってしまわれた …… orz そのまま落ち込んでいても液晶は勝手に直ってくれるわけではないので、Apple のサポートページやらそれっぽいところをぐるぐると見ていると、なんと、持ち込みで修理ができる Genius Bar なるものがあるらしい。なんでもこの Genius Bar、Apple 製品についての幅広い知識を持つスタッフがマンツーマンで動作の不具合や故障だけでなく、Apple 製品の使い方や、利用上の相談にも乗ってくれるところらしい。すぐさま翌日の予約を iPhone の Apple Store アプリを用いて取り付けます。予約ではどの商品についてのサポートを受けるのか、日時、場所、サポートの希望内容などを尋ねられるので、それぞれ、Mac、翌日の 18:00、Apple Store Shibuya、液晶の不具合として予約。そして翌日、念のために Time Machine でバックアップをとった後、残念液晶となった MacBook を抱えて Apple Store Shibuya 内の Genius Bar を目指すのでした ……

2 Genius Bar にて

JR 渋谷駅から徒歩五分程、Apple Store Shibuya の二階に Genius Bar はある。まず、Genius Bar に入ってから入り口近くに立っていたスタッフに声をかけて予約を確認してもらい、順番が来るまで中を見まわりながら待ちます。Genius Bar の中は大変な混みようで、チェックインしてから五分程度で順番が回ってきたけれども、予約して来なかったらいったいどれほど時間が掛かっていたのだろうか …… 。

予約の時点でサポート希望内容は伝えてあるので話の流れはスムーズでした。実際の症状の確認、経緯、スクリーンショットを撮ってみてハード側に問題があることを自身で確認したことなどを伝えると、Genius[*1]は別のモニタに私の MacBook から出力して確認し、これに問題がないことを確認、症状を診る限り液晶ユニットでの電気的な問題があると予想されるとのこと。そのまま話は修理の見積もりへ。今回は液晶ユニットを丸ごと交換することになったので液晶ユニット代と修理の技術料が通常かかるものの、購入から一年以内の保証期間内[*2]ということで修理は無料で行える運びとなりました。修理には一週間ほど掛かる

[*1] Genius Bar で Apple 製品についての幅広い知識を持つ人をこう呼ぶらしい。

[*2] 保証期間であることを証明するような書類などは必要ない。シリアルナンバーから確認してもらえる。

とのことで、修理完了のお知らせを電話かメールでもらえるとの事なので、メールでもらうことにし、修理確認書類にサインをして MacBook を預けてその日は帰ったのでした。

3　直っ … た ……?

修理に出してから三日目の午後、知らない番号から電話が掛かってきたので出てみると、Genius Bar からの電話でした。その一部始終は以下。[*3]

Genius「こちら Apple Store Shibuya、Genius Bar の○○と申します。」

おお、一週間掛かるという話だったけれどもう直ったのか。しかし、メールで修理完了の知らせをもらうはずだったのだが ……?

Genius「お預かりしていた MacBook Pro 15in の液晶ユニットの交換は終わったのですが、」

おい、ちょっと待て。「が」ってなんだ。「が」って。

Genius「Mac が正常に立ち上がらなくなってしまいまして、SSD に問題が生じているのですが、こちらのほうで換装してもよろしいですか？もちろんご料金はいただきません。」

はい …? 今ナントオッシャイマシタカ？コノヒトハ？

あまりのことに何も考えられずに「はい、お願いします」と返していた私がそこにいた。

Genius「SSD の換装の際にはデータがすべて消去されて工場出荷時の状態に戻ってしまうのですが、データのバックアップなどはお取りしておられますか？」

私「はい、取ってあります …… 」

Genius「では、こちらで SSD の換装を行わさせて頂きます。SSD はこれから発注致しますので、申し訳ございませんが、このあと 3 日ほどお時間を頂けますか？」

私「はい … 。」

と、まぁ、そのあと適当にやり取りがあって電話は切れたわけですが。メールじゃなくて電話が掛かってきた時点で変だなぁとは思ったけれども、まさか修理に出して液晶が直ったと思ったら今度は SSD がお亡くなりになるとは … 誰が予想出来たでしょうね？いや、もうどう考えても修理中に SSD が Genius の手によって破壊されてるじゃないてすかー。ホントなにしてくれちゃってるんですかー？もう丸ごと新品交換のほうが早いじゃないですかー。環境戻すの手間じゃないですかー。などと恨みがましく思いもしたものの、ゴミファイルの整理になると前向きに考えることにしました。バックアップ取っといてホント良かったです、はい … 。

4　修理完了とそれから

SSD 破壊の報告を受けてから三日後、今度こそ修理が完了した旨がメールで届いたので、その翌日に MacBook を受け取りに再び渋谷へ。今回は受け取るだけなので予約は要らないだろうと思い、そのまま向かいました。到着後、受け取りに来た旨をスタッフに伝え、待つこと十分ほど。順番が回って来てのでカウンターへ受け取りに向かうと Genius が奥から修理の完了した MacBook を持ってきました。受け取り時は液晶その他の確認を促されたのですが、ここで一つ驚いたことが。なんと私の MacBook の外装が新品に交

[*3] 確かこんな感じだった気がする。

換されていたのです。SSD は破壊されたものの、Genius Bar に修理に出して良かったと思った瞬間でしたね。なお、修理時の最終的な見積もりは、次のようになっていました。

Display Clamshell, Grossy（液晶ユニット）	￥39,491
Hard Drive, 128 GB, SSD, 2.5 inch	￥52,700
Hardware Repair Labor（技術料）	￥3,250
合計	￥95,441

修理費合計 95,441 円って … 、MacBook Air とか買えちゃいますね。保証期間内で本当に良かったです、はい。ちなみに本当に工場出荷時の状態で戻って来たので、Mountain Lion が Lion に戻っていましたね。削除されたデータについては、ゴミファイルの掃除を兼ねて、データはバックアップから必要なものだけちまちま取り出す作業、ソフトウェアは入れなおすという形をとりました。そして修理が完了した MacBook を受け取ってから三日後、Apple から「最近ご利用いただいたアップル・ストアでの修理について」という修理のサービスについてのアンケートが来ました。適当に答えていくとこんな質問が、

「修理後の MacBook Pro の外観はどうでしたか？」

この質問を目にしたときは思わず吹き出しそうになりましたが、こんな質問があるということは Genius Bar に修理に持っていくと外装が新品になって戻ってくるという認識でいいのでしょうか？

5　終わりに

液晶直しに行っただけなのに思わぬ不幸（幸？）に見舞われた Genius Bar でしたが、Apple 製品を持っている方は何かあった際には利用を検討してみても良いかと。ここであえて「お勧め」と書かない理由は …… まあ、こういうこともありますよ、という話です。ああ、あと修理前に焦らないために、データのバックアップは定期的に取るようにしましょう。以上、Genius Bar 利用レポートでした。

キーボード マニア

mznh

はじめに
アメリカ西部のカウボーイたちはなんとかかんとか、というわけで最近買ったキーボードを独断と偏見に基づいてレビュー

1 簡単な説明

レビューをする前にキーボードに関するいくつかの単語をざっくりと説明しよう。

1.1 メンブレン式

デスクトップ PC とかを買ったらついてくるキーボードはまず間違いなくこの方式のキーボードだろう。ラバーキャップが中に入っていてそれが凹んで中の基盤と接触し通電することにより入力を感知する。安価なのが特徴。その反面、ラバーキャップが劣化したりと耐久性に欠点がある。

1.2 メカニカルスイッチ式

その名の通りキーのひとつひとつにマイクロスイッチが入っていてそれが押されることにより入力を感知する。使われるマイクロスイッチの違いによって赤軸とか青軸とか言ったりする。赤軸は静かさ、青軸はクリック感、黒軸は反発力、などなどスイッチ毎に特徴があり好みがわかれるところ。メンブレン式と比べ安くとも 5 千円で 1 万超の物もあり高価、また本体重量が重い。

1.3 静電容量無接点

電子部品のコンデンサーのような構造になっていてキーが押下されることにより静電容量が変化、それを検出し入力を感知する。物理的なスイッチを持たないため耐久性が高く音も静か。しかし価格は 2 万円前後が主流と他と比べ高価。

1.4 パンタグラフ式

最近は後述のアイソレーションキーボードに押され気味だがノート PC のキーボードと言ったらこれ、由来は内部でキートップを支えている構造が電車のパンタグラフに似ているため。薄いのが特徴。

1.5 アイソレーションキーボード

キーとキーが 2~ 3mm 程離れているキーボード、最近のノート PC はこれを採用しているものが多い。爪が伸びていても前のキーに引っかからなかったりするのが利点らしい。（筆者はあまり好きではない）

1.6 キーストローク

キーを押下した時に沈む深さ。これが大きいとしっかりした打心地を得られる、逆に小さいと軽いタッチで打てる。

1.7 キーピッチ

キー自体の大きさのこと。大きいと打ちやすいが小さいと小型化出来る。

2 PFU 製 Happy Hacking Keyboard Lite2 for mac

製造元	株式会社 PFU
種類	メンブレンスイッチ
キー配列	JIS 配列 68 キー

HappyHacking シリーズの廉価版、所謂情報基盤センターのキーボード。秋葉原でワゴンに放り込まれてたものを 3k 程で購入。中古故か野ざらし故か若干黄ばんでいる。メンブレン式ながらキータッチは軽くストロークも深く打心地は良好。打鍵音もポコポコと楽しげで良い。軽くコンパクトなので持ち運びも可能。

打心地	★★★☆☆
デザイン	★★★★☆
打鍵音	ポココポココココ …
情基度	★★★★☆

3 Owltech 製 LED LIGHT KEYBOARD STELLAR109

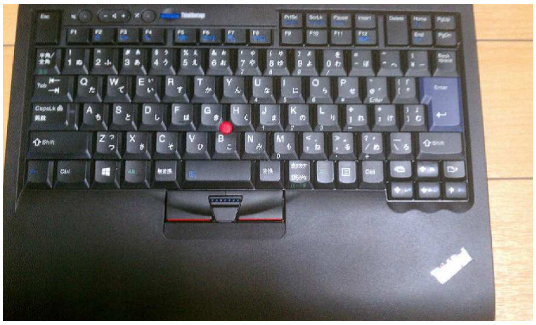
製造元	株式会社 Owltech
種類	メカニカルスイッチ (青軸)
キー配列	JIS 日本語 109

秋葉原の某 99 で一目惚れし購入、価格は 13k 程だった（後日 Amazon で安く買える事を知る）。採用されているスイッチになぞらえてか LED によりキーが青く光るのが最大の特徴。青軸らしくキータッチは軽く打鍵音はカチャカチャと賑やか、打鍵音に合わせて伝わるクリック感はまるで霜柱の上を歩いているかのよう。打鍵音に LED と派手なキーボードだ。

打心地	★★★★☆
デザイン	★★★☆☆
打鍵音	カチャチャチャチャ…
うるささ	★★★★★

4 lenovo 製 ThinkPad USB トラックポイントキーボード

製造元	レノボ・ジャパン社
種類	パンタグラフ式
キー配列	日本語/4 ホットキー



ThinkPad のキーボードをそのまま取り外したキーボード、真ん中にトラックポイントがついておりポインティングデバイスも兼ねている。パンタグラフ式なため本体が他の外付けキーボードと比べ非常に薄く軽い、そのため持ち運びも用意なのだがこれを購入する人は大抵 ThinkPad も持っているだろうからこのキーボードを持ち運ぶことはないだろう。打心地はパンタグラフにしては深めのキーストロークで流石 ThinkPad といえる打ちやすさ、打鍵音は低め。トラックポイントも一度慣れてしまうとキーボードから手

を離さずにポインティングができるので快適。問題点はトラックポイントのドライバが本家 ThinkPad とすこし違った挙動をするところか。

打心地	★★★★☆
デザイン	★★★☆☆
打鍵音	ダカダカカダカカ …
TKB 度	★★★★★

5　最後に

キーボードマニアなんてタイトルにしておきながら「種類少ないな」、とか「静電容量無接点ないのかよ」とか思われてらっしゃるかもしれませんが予算の都合です。あと店頭で触った感触だけで LaVie Z のキーボードをこき下ろそうかと思いましたが流石に持ってないキーボードを酷評するのはアレだなと思ったのでやめました。

次があったら青軸以外のメカニカルもレビューしようかと思ってます。

つくる! WM

clear

はじめに

2012 年秋号でウィンドウマネージャ(WM) を自作した話を書きました[*1]が、時間や紙面の不足から実際のプログラムについて全く触れられませんでした。本稿では、秋号で触れられなかった WM の内部についてコード例を示しつつ述べます。

1 WM は何をするのか

WM は X Window System[*2]においてウィンドウの操作体系や装飾を提供するプログラムです。世の中には星の数ほどの WM が存在し、どのような WM を使うかによってデスクトップの見た目や操作感が大きく変わってきます。自分に合った WM を選択することで、作業効率が向上して色々と捗るかもしれません。

WM の大きな仕事は、ウィンドウの操作と、操作のためのインターフェースの提供です。

1.1 ウィンドウの操作

X では、WM が存在しない限りウィンドウを移動したり、リサイズすることはできません。ウィンドウの主であるプログラムが必要に迫られてウィンドウの移動やリサイズを行った場合は別ですが、そのようなことは基本的には WM に任せるべきであるとされています。

1.2 インターフェースの提供

WM が存在しない場合、ウィンドウにタイトルバーが付いたり、ユーザがドラッグして引っ張ることでウィンドウサイズを変更できる枠が付いたりすることはありません。このようなウィンドウの装飾は WM に任されており、そのために WM の違いは見た目の違いに直結します。見た目だけでなく、ウィンドウの操作体系も WM によって異なってきます。

WM がインターフェースを提供する相手はユーザのみに限りません。ウィンドウの主であるプログラムがウィンドウの移動やリサイズを要求することもあります。要求をどう扱うかは WM 次第で、希望通りにする場合もあれば、一部の要求に関しては調整を行ったり、あるいは完全に要求を無視することもあります。また、WM ともウィンドウの作成者とも異なる外部のプログラム、例えばタスクバーなどからウィンドウの操作要求 (「最小化」など) が送られてくる場合もあり、これに対応するのも WM の仕事です[*3]。

[*1] 過去の百萬石は `http://wiki.mma.club.uec.ac.jp/Booklet` にあります

[*2] X とか X11 などとも呼ばれる、Unix 系 OS において用いられているウィンドウシステム

[*3] X ではデスクトップの構成要素それぞれが独立したプログラムであり (複数の要素を兼ね備えるものもある)、色々と組み合わせて使うことができたりできなかったりする

2 XCB

X はクライアントサーバモデルを採用しており、各種の操作は X サーバに対してリクエストを送ることで達成されます。X 上で動作するプログラムを作成する時は何らかの GUI ツールキット (GTK+、Qt など) を使うのが一般的ですが、WM を実装する際にはツールキットが提供しない低層の処理を行う必要が出てきます。X プロトコルを直接扱うためのライブラリとして、ここでは XCB(X C Bindings) を用います。これは名前の通り C 向けのライブラリで、旧来用いられてきた Xlib よりもダイレクトに X プロトコルを扱える点が特徴です。適切に扱えばパフォーマンスは Xlib を上回ります。

2.1 XCB の基礎

XCB については資料が非常に少ないので、紙面を割いて簡単に紹介しておきます。XCB は X クライアントを作成するためのライブラリで、次のようなことを可能にします。

- リクエストの送信
- イベント/リプライ/エラーの取得

X サーバとの間の通信は単なるソケット通信[*4]なので、完全に自前で X プロトコルをしゃべることができるのであれば XCB は不要です (非現実的ですが)。XCB は X プロトコルをしゃべるためのライブラリであるともいえます。

先に述べた通り、やりたいことを実現するためには X サーバにリクエストを送る必要があります。このリクエストは大きく二種類に分かれます。

- リクエストのみを行うもの
- リクエストに対するリプライを要求するもの

リクエストのみを行うものには、例えば「このウィンドウを指定した座標に移動してくれ」とか「このウィンドウのこの属性を変更してくれ」といったものが含まれます。これは単純に要求のみを行うもので、結果は成功したか失敗したか、それだけです。一方、リプライを要求するものには、例えば「ウィンドウの一覧をくれ」とか「このウィンドウがどの座標にどんなサイズで存在するか教えてくれ」といったものが含まれます。これらについては、成功/失敗の他、成功した場合は要求した情報がリプライとして送られてきます。

さらに、エラー処理についても 2 通りが考えられます。リクエストの成否 (と、失敗した場合どのようなエラーが発生したか) がすぐに必要な場合、言い換えれば失敗した場合特別に処理が必要な場合と、すぐには必要ない場合、すなわち失敗したら失敗したで後でそのことが分かれば良い (あるいは、分からなくても良い) 場合です。

リクエストがリプライを要求するものであるかどうかと、エラーをその場で取得するかどうかの組み合わせは 4 通り存在し、XCB ではこの 4 通りの方法を任意に選択して用いることができるようになっています。

ここから先は、実際にコードを見ながら解説してゆきます。

[*4] ネットワーク越しなら TCP/IP、サーバとクライアントが同一ホスト上にあるなら UNIX ドメインソケット

3 何もしない WM

本稿を執筆するにあたって題材とする WM を考えたところ、まともな WM は前提とする知識[*5]が多く、題材として現実的でないという問題に行き当たりました。そこで、本稿では「何もしない」WM を題材として WM の内部について述べたいと思います。「何もしない」とは「外から見て WM が存在しないのと変わらないような挙動を取る」ということです。その他の唯一の機能として、「他の WM が起動できなくなる」というものがあります。普通 WM は同時に 1 つしか起動することができないようになっています[*6]。

大した分量はないので、初めにプログラムの全体を示します[*7]。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <xcb/xcb.h>
4  #include <xcb/xcb_aux.h>
5  #include <xcb/xcb_event.h>
6
7  xcb_connection_t *conn;
8  xcb_screen_t *screen;
9
10 void configure_window(xcb_configure_request_event_t *e)
11 {
12     uint32_t values[7];
13     int i = 0;
14     if (e->value_mask & XCB_CONFIG_WINDOW_X)
15         values[i++] = e->x;
16     if (e->value_mask & XCB_CONFIG_WINDOW_Y)
17         values[i++] = e->y;
18     if (e->value_mask & XCB_CONFIG_WINDOW_WIDTH)
19         values[i++] = e->width;
20     if (e->value_mask & XCB_CONFIG_WINDOW_HEIGHT)
21         values[i++] = e->height;
22     if (e->value_mask & XCB_CONFIG_WINDOW_BORDER_WIDTH)
23         values[i++] = e->border_width;
24     if (e->value_mask & XCB_CONFIG_WINDOW_SIBLING)
25         values[i++] = e->sibling;
26     if (e->value_mask & XCB_CONFIG_WINDOW_STACK_MODE)
27         values[i++] = e->stack_mode;
28     xcb_configure_window(conn, e->window, e->value_mask, values);
29 }
30
31 int main(void)
32 {
33     int screen_num;
34     conn = xcb_connect(NULL, &screen_num);
35     if (xcb_connection_has_error(conn)) {
36         fputs("can't open display\n", stderr);
37         exit(1);
38     }
39
40     screen = xcb_aux_get_screen(conn, screen_num);
41     xcb_void_cookie_t cookie = xcb_change_window_attributes_checked(
42         conn, screen->root, XCB_CW_EVENT_MASK,
```

*5 X の諸概念に関するものと、クライアント間通信に関する規約

*6 より正確には、各ルートウィンドウに対して 1 つ

*7 ソースコードは http://www.mma.club.uec.ac.jp/~clear/ から

```
43          &(const uint32_t){ XCB_EVENT_MASK_SUBSTRUCTURE_REDIRECT });
44      xcb_generic_error_t *e = xcb_request_check(conn, cookie);
45      if (e) {
46          fputs("another window manager is running\n", stderr);
47          exit(1);
48      }
49
50      xcb_generic_event_t *event;
51      while ((event = xcb_wait_for_event(conn))) {
52          if (event->response_type == 0) {
53              xcb_generic_error_t *e = (xcb_generic_error_t *)event;
54              fprintf(stderr, "X protocol error: request=%s, error=%s\n",
55                  xcb_event_get_request_label(e->major_code),
56                  xcb_event_get_error_label(e->error_code));
57              exit(1);
58          } else {
59              switch (XCB_EVENT_RESPONSE_TYPE(event)) {
60              case XCB_MAP_REQUEST:
61                  xcb_map_window(conn,
62                      ((xcb_map_request_event_t *)event)->window);
63                  break;
64              case XCB_CONFIGURE_REQUEST:
65                  configure_window((void *)event);
66                  break;
67              }
68              xcb_flush(conn);
69          }
70          free(event);
71      }
72      if (xcb_connection_has_error(conn))
73          fputs("connection with X server closed\n", stderr);
74      else
75          xcb_disconnect(conn);
76      return 0;
77  }
```

はじめに configure_window というなんだかよく分からない関数がありますが、ひとまずこれは置いておいて、main の方から見ていきます。

3.1 X サーバへの接続

まずは X サーバに接続します。

```
33  int screen_num;
34  conn = xcb_connect(NULL, &screen_num);
35  if (xcb_connection_has_error(conn)) {
36      fputs("can't open display\n", stderr);
37      exit(1);
38  }
```

xcb_connect で X サーバへの接続を行います。この関数は xcb_connection_t 型のポインタを返しますが、これは以降 X サーバと何らかのやり取りをするときは必ず必要になります。接続に成功したかどうかは xcb_connection_has_error で分かります。これが 0 以外の値を返した場合何らかのエラーが発生しています。

3.2 ルートウィンドウの取得とイベントマスクの設定

普通のクライアントプログアムであれば、X サーバに接続した後は自分のウィンドウを作成して初期化処理などを行うところですが、WM は特定のウィンドウ上だけで何かする訳ではないので、ここではウィンドウを作成することはしません。代わりに、全てのウィンドウの祖先にあたるルートウィンドウを監視します[*8]。ルートウィンドウは画面全体をカバーするウィンドウで、大きさは画面の大きさと等しくなっています。

全てのウィンドウは固有の ID を持っており、ルートウィンドウも例外ではありません。ウィンドウに対して操作を行う際にはその ID をもってウィンドウを指定します。ここでは、ルートウィンドウのイベントマスクという属性を変更することによって、ルートウィンドウ上でイベントが発生したときにそれが通知されるようにします。

```
40 screen = xcb_aux_get_screen(conn, screen_num);
41 xcb_void_cookie_t cookie = xcb_change_window_attributes_checked(
42     conn, screen->root, XCB_CW_EVENT_MASK,
43     &(const uint32_t){ XCB_EVENT_MASK_SUBSTRUCTURE_REDIRECT });
44 xcb_generic_error_t *e = xcb_request_check(conn, cookie);
45 if (e) {
46     fputs("another window manager is running\n", stderr);
47     exit(1);
48 }
```

ルートウィンドウの ID は `xcb_screen_t` 構造体の `root` メンバで分かります。`xcb_aux_get_screen` という関数によってこの構造体へのポインタが得られます。

ID が分かったら、ルートウィンドウのイベントマスクを変更します。X プロトコルではウィンドウの属性を変更する ChangeWindowAttributes というリクエストが定められているのでこれを送れば良いのですが、ここで先に述べた 4 通りの方法が問題となります。まず、このリクエストはリプライを要求するものではないので、2 通りまで絞りこめます。あとはエラー取得のタイミングです。大抵のリクエストであれば投げっ放しで良いのですが、今回は成否が重要なのでその場でエラーを取得します。理由は設定しようとしているイベントマスクの値にあります。

イベントマスクは、そのウィンドウに関してどのようなイベントを通知して欲しいかを指定する値です。マスクとある通り複数の値を組み合わせて指定し、その一つ一つがイベントの集合に対応しています (マスクとイベントは一対多の対応)。今回指定するのは SubstructureRedirect という値で、MapRequest と ConfigureRequest という 2 つのイベントに対応しています。これらについては後ほど詳しく述べます。さて、イベントマスクの値はプログラムごとに独立した値を設定できます。そのため、複数のクライアントが同じウィンドウを監視するといったことも可能ですが、SubstructureRedirect に関しては同時に 1 つのクライアントしか設定することができないようになっています。「WM を同時に複数動かすことはできない」のはこのことに起因しています。普通の WM はルートウィンドウのイベントマスクに SubstructureRedirect を設定するので、既に WM が動いていれば後から行われたリクエストは失敗します。要するに、「既に他の WM が動作していないか」がリクエストの成否によって確認できるということです。

[*8] ウィンドウの親子関係は木構造となっている

XCB においては、リクエストとそれを X サーバに送信する関数が一対一対応しています。ChangeWindowAttributes であれば xcb_change_window_attributes が対応します。対応する関数を呼んでリクエストを発行すると、クッキー (cookie) という値が返ってきます。これは主にリプライを必要とするリクエストを送る際に重要で、リクエストの種類に応じた型のクッキーが存在します。XCB においてはリクエストの発行とリプライの取得はそれぞれ別々に実行でき[*9]、複数のリクエスト/リプライをやり取りする際は初めに全てのリクエストを送信してから順次リプライを取得すると効率が良いとされています。このような場合に、クッキーが個々のリクエストとリプライを結びつける目印となります。

リプライを要求しないリクエストに関しては、対応するクッキーの型は xcb_void_cookie_t となっており、今回呼んでいる xcb_change_window_attributes_checked もこれを返します。関数名の末尾に _checked と付いていますが、これがエラーをその場で取得する方法を取ることを示しています。_checked と付いている関数は、リクエストを送信した後その結果が判明するまでブロックします。付いていない方の関数も存在し、こちらはブロックしません。さて、返ってきたクッキーを xcb_request_check に渡すとエラーを取得できます。エラーがあれば xcb_generic_error_t 構造体への有効なポインタが返され、エラーがなければ NULL が返ってきます。xcb_request_check を使うためには _checked の方の関数を使ってリクエストを送信する必要があります。もしエラーが発生していたら、既に他の WM が存在しているので直ちに終了します。

3.3 イベントループ

イベントマスクを設定したことにより、ルートウィンドウ上でイベントが発生するとそれが届けられてくるようになります。これを監視し処理するのがイベントループです。

```
50 xcb_generic_event_t *event;
51 while ((event = xcb_wait_for_event(conn))) {
52     if (event->response_type == 0) {
53         xcb_generic_error_t *e = (xcb_generic_error_t *)event;
54         fprintf(stderr, "X protocol error: request=%s, error=%s\n",
55                 xcb_event_get_request_label(e->major_code),
56                 xcb_event_get_error_label(e->error_code));
57         exit(1);
58     } else {
59         switch (XCB_EVENT_RESPONSE_TYPE(event)) {
60         case XCB_MAP_REQUEST:
61             xcb_map_window(conn, ((xcb_map_request_event_t *)event)->window);
62             break;
63         case XCB_CONFIGURE_REQUEST:
64             configure_window((void *)event);
65             break;
66         }
67         xcb_flush(conn);
68     }
69     free(event);
70 }
```

xcb_wait_for_event でイベントを取得します。この関数はイベントが来るまでブロックし、届いたものを返してきます。各イベントにはそれぞれ対応する構造体が存在しますが、「全てのイベント」を表すものとし

[*9] Xlib においては不可分となっている

て xcb_generic_event_t 構造体が用意されており、一つの型で全てのイベントを受け取ることが可能になっています。また、xcb_generic_event_t へのポインタを具体的なイベント構造体へのポインタにキャストできるように設計されています。X サーバとの通信で入出力エラーが起こった場合は NULL が返ってきます。よって、このループは入出力エラーが発生するまで永遠に続くことになります。また、xcb_wait_for_event が返してくるポインタは呼び出し側の責任で free しなければなりません。

xcb_wait_for_event はイベントだけでなくエラーを返してくることもあります。エラーの場合、xcb_generic_event_t 構造体が実際にはどのイベントであるかを示す response_type メンバが 0 になります。リクエストを送信する際に「エラーは後で分かれば良い」方法をとった場合、発生したエラーはこのタイミングでやってきます。今回は、エラーが来た場合はそれがどのリクエストによるものであるのか、またどんなエラーなのかを表示した後、単に終了しています。いきなり終了してしまうのはあまり良くないのですが、今回は扱うリクエストがごく限られており、エラーは通常発生しないであろうと期待できるので楽をしています。

届いたのがイベントであった場合は、イベントの種類に応じた処理を行います。

3.4 イベントのリダイレクト

今回処理する 2 つのリクエストの名前には共に Request と付いています。これは、これらのイベントがリクエストに付随して発生するものであることを示しています。

- MapRequest: ウィンドウの表示を要求する MapWindow リクエストに対応
- ConfigureRequest: ウィンドウの座標やサイズの変更を要求する ConfigureWindow リクエストに対応

これら 2 つのリクエストは、いずれもウィンドウの配置を左右するリクエストです。(WM 以外の) クライアントがこれらのリクエストを自由に使えると、クライアントが WM の支配を逃れて自由にウィンドウを配置できることになり、WM が十分に役割を果たせません。そこで、リクエストが直接 X サーバではなく WM の所に送られてくる (リダイレクトされる) ようにする仕組みが用意されています。具体的には、あるウィンドウの親ウィンドウのイベントマスクに SubstructureRedirect を設定したクライアントが存在する場合、MapRequest と ConfigureRequest はそのクライアントにリダイレクトされます。WM が管理するのはルートウィンドウの直接の子であるウィンドウ (トップレベルウィンドウ) なので[*10]、ルートウィンドウで SubstructureRedirect を設定しておけば、WM の管理対象であるトップレベルウィンドウからのリクエストを得ることができます。

リダイレクトされたリクエストは、WM が何もしなければそのまま闇に葬られます。一般的な WM は自らのポリシーに従ってクライアントからのリクエストを改変して X サーバに送信します。状況その他の要因によって、全く手を加えずに受け流す場合もあれば、逆に完全に無視して闇に葬る場合もあります。「何もしない WM」は WM が存在しないかのように振る舞う必要があるので、クライアントからのリクエストをそのままの形で X サーバに送ればよい、ということになります。

[*10] ウィンドウ内のボタンやテキストボックスなどはそれぞれがウィンドウであり、トップレベルウィンドウの子として作られるが、これらについて WM は一切感知しない

それでは、それぞれのリクエストについて見て行きます。

3.5 MapRequest

MapRequest はウィンドウの表示を要求する MapWindow リクエストがリダイレクトされてきたものです。ウィンドウは作成された時点では画面上に表示されず、MapWindow によって画面上に表示されるようになります。「何もしない WM」としては、要求通りにウィンドウを表示すれば良いでしょう。

```
60 case XCB_MAP_REQUEST:
61     xcb_map_window(conn, ((xcb_map_request_event_t *)event)->window);
62     break;
```

MapWindow をそのまま発行してやればよいので、対応する関数 `xcb_map_window` を呼び出しています。第 2 引数に表示するウィンドウの ID を指定するのですが、これはイベント構造体の中に入っています。MapRequest に対応するのは `xcb_map_request_event_t` で、この中の `window` メンバが表示するウィンドウです。クライアントが変なリクエストを送っていない限りエラーは発生しないはずであり、またエラーを取得してもあまり意味がないので、`_checked` の付かない方の関数を用いています。こちらの方がリクエストを投げた後ブロックしない分高速であり、普通リクエストを投げるときはこちらを用います。仮にエラーが発生した場合はイベントループで検出されます。

3.6 ConfigureRequest

ConfigureRequest は ConfigureWindow リクエストがリダイレクトされてきたものです。ConfigureWindow リクエストは、ウィンドウの以下のパラメータを変更します。

- ウィンドウの座標 (ウィンドウを囲む矩形の左上の角の座標で表される)
- ウィンドウのサイズ
- ウィンドウ枠の太さ
- ウィンドウの上下関係 (いわゆる Z オーダー)

「何もしない WM」では、MapWindow の場合と同じく要求通りにリクエストを再発行します。これを行うのがプログラム中の `configure_window` 関数です。

```
10 void configure_window(xcb_configure_request_event_t *e)
11 {
12     uint32_t values[7];
13     int i = 0;
14     if (e->value_mask & XCB_CONFIG_WINDOW_X)
15         values[i++] = e->x;
16     if (e->value_mask & XCB_CONFIG_WINDOW_Y)
17         values[i++] = e->y;
18     if (e->value_mask & XCB_CONFIG_WINDOW_WIDTH)
19         values[i++] = e->width;
20     if (e->value_mask & XCB_CONFIG_WINDOW_HEIGHT)
21         values[i++] = e->height;
22     if (e->value_mask & XCB_CONFIG_WINDOW_BORDER_WIDTH)
23         values[i++] = e->border_width;
24     if (e->value_mask & XCB_CONFIG_WINDOW_SIBLING)
25         values[i++] = e->sibling;
```

```
26     if (e->value_mask & XCB_CONFIG_WINDOW_STACK_MODE)
27         values[i++] = e->stack_mode;
28     xcb_configure_window(conn, e->window, e->value_mask, values);
29 }
```

ConfigureWindow リクエストに対応するのは xcb_configure_window です。この関数は第 3 引数にどのパラメータを変更するかのマスクを取り、第 4 引数にパラメータの値の配列を取ります。これはただの配列なので順番が重要で、マスク値が定義されている順に入れて行かなければなりません[*11]。そのため上のような書き方をしています。

3.7 イベントの送信

リクエストに対応する関数を呼び出しても、X サーバにすぐにリクエストが送信されることはありません。リクエストは XCB のバッファにたまって行き、以下の条件のいずれかが満たされると送信されます[*12]。

- リプライを取得する関数が呼ばれる。あるいは、xcb_request_check が呼ばれる
- バッファが埋まる
- xcb_flush が呼ばれる

リクエストが確実に送信されるようにするためには、xcb_flush を呼んで明示的にバッファをフラッシュしてやる必要があります。これを忘れるといつまでたってもリクエストが処理されずに悩むことになるので注意が必要です[*13]。プログラム中では、イベント処理の後に xcb_flush を呼んでいます。

3.8 終了処理

イベントループを抜けたところでもう一度 xcb_connection_has_error によってエラーの有無を確認しています。

```
72 if (xcb_connection_has_error(conn))
73     fputs("connection with X server closed\n", stderr);
74 else
75     xcb_disconnect(conn);
```

実はこの if 文には意味がありません。というのも、イベントループを抜ける条件が入出力エラーの発生なので、条件が必ず真になるからです[*14]。ですが、正常終了するときの処理を示すためにあえてそのままにしています[*15]。xcb_disconnect は X サーバとの接続を切る関数です。

[*11] XCB のヘッダファイル xproto.h を見ると分かる
[*12] http://lists.freedesktop.org/archives/xcb/2008-December/004152.html による
[*13] 経験者は語る
[*14] 書いてから気づいた
[*15] 適当な条件でイベントループを break するようにした場合、ループを抜けた原因を知るために必要となる

4 この先

大変残念ながら「何もしない WM」は全く何の役にも立ちません。これだけで終わるのも悲しいので、その先の展望として、まともな WM が実装すべき事柄を挙げてみます。

- ウィンドウの情報の管理
- ウィンドウ配置の制御
- 入力フォーカスの管理
- ウィンドウの装飾
- ユーザによる操作の処理
- クライアント間通信への対応

4.1 ウィンドウの情報の管理

ほぼ全ての WM が、自分が管理対象としているウィンドウの情報を保持しています。基本的なものではウィンドウの ID や座標、サイズなどがあり、実用的な WM を目指すならばウィンドウのプロパティなどさらに多くの情報を扱う必要があります。ウィンドウを表す構造体やクラスを用意して、そのリストを保持するのが一般的です。リストへの追加は MapRequest イベントの時点で行い、ウィンドウが非表示になったり (UnmapNotify イベント)、破棄された時点 (DestroyNotify イベント) でリストからの削除を行います。

4.2 ウィンドウ配置の制御

これは WM の最も基本的な機能です。ユーザの操作に応じて移動・リサイズしたり、ウィンドウが画面の範囲外に飛び出して行かないように見守ったり、ウィンドウ間の上下関係を適切に管理したり、マルチモニタ環境で適切にウィンドウが配置されるように配慮したりと、考えることは山ほどあります。

4.3 入力フォーカスの管理

フォーカスとはユーザの入力を受け取る権利であり、これはある意味で限られた資源だともいえます。ユーザの操作やその時々の状況に応じて適切なウィンドウにフォーカスを与えるのが WM の使命です。フォーカスの扱いを誤ると深刻な問題 (例えば、完全に操作不能な状況) が発生しうるので注意が必要です。

4.4 ウィンドウの装飾

ウィンドウにタイトルバーをつけてアイコンやタイトルを表示したり、「閉じる」「最小化」などのボタンを付けたりするのは WM の仕事です。最低限の枠しか付けない WM から派手な装飾を施す WM まで様々で、見た目に関わってくるだけでなく、どのような操作体系を可能とするかにも関わってきます。

4.5 ユーザによる操作の処理

これが無くては話になりません。最も基本的なマウスを使ったウィンドウの移動・リサイズから、キーの組み合わせによって特定の機能を実行するような機能まで、WM の使い勝手にダイレクトに関わってくる部分です。

4.6 クライアント間通信への対応

外部のクライアントが WM に対して何らかの要求をしてくることがあります。典型的なのはタスクバーとかパネルと呼ばれる類のものでしょう。このようなクライアントは WM ではないのにも関わらず、これらを通してウィンドウを操作 (例えば、最小化とそこからの復帰) することができます。これは外部のクライアントが WM へ要求を送り、WM がそれを解釈して実行するという一種のクライアント間通信によって達成されています。

また、ウィンドウの配置は WM に任されているとはいえ、ウィンドウの主であるクライアントからウィンドウの配置方法について WM に注文をしたい場合も考えられます。典型的な例としては、固定サイズウィンドウや、最大・最小サイズが定まっているウィンドウなどが挙げられます。このような場合「サイズヒント」と呼ばれるプロパティをウィンドウに設定しておくことで、WM に情報を提供することができます。

WM と外部のクライアントがやりとりをするための規約は以下の 2 つの文書で定められています。

- Inter-Client Communication Conventions Manual(ICCCM)[*16]
- Extended Window Manager Hints(EWMH)[*17]

これらの規約に従うと、どこかの誰かが書いたクライアントとも連携することが可能になります[*18]。

5 おわりに

以上、長々とお付き合い下さりありがとうございました。本当は私の自作 WM(秋号のときに Xlib を使って作ったものを`/dev/null` に放り込んだ後、XCB を使って一から書きなおしたもの) について語りたかったのですが、WM としての完成度がまだまだ不十分な上に語ろうとすると前提知識を要求しすぎるので、このような形となりました。それでも XCB に関する説明にだいぶ紙面を割いてしまった感はあります。

WM を作ってみたいという奇特な方には[*19]、私からは既存の WM、それもなるべく小さいものを読んで改造するところから始めることをおすすめします。いきなりフルスクラッチで書こうとしても右も左も分からない上、使い物になるまでの期間が長すぎて間違いなく挫折します。WM 開発には楽しい部分と創造性の欠片もない面倒な部分[*20]とが混在しているところがあるので特に。evilwm あたりが規模と機能のバランスが取れていて良いかと思います。それから、いきなり XCB を使うのはおすすめしません。XCB は極端にドキュメントが少なく[*21]、また XCB を用いて実装された WM もそれほど多くはないのが現状です。旧来の Xlib をある程度知った所から XCB に移行するのはそれほど難しくなく、また Xlib で書かれた WM の方が圧倒的に多いので、初めは Xlib に触れてみることをおすすめします (ただ、Xlib や XCB は WM を書くのでもない限りほとんど使う必要がないので、GTK+ や Qt を使えるようになった方が間違いなく有益です)。

[*16] `http://www.x.org/docs/ICCCM/icccm.pdf`
[*17] `http://standards.freedesktop.org/wm-spec/wm-spec-latest.html`
[*18] 残念ながら現実はそれほど優しくなく、規約を把握して対応すること自体が大変な上、クライアントによって微妙に解釈が違ったりもして割とつらい
[*19] もしいたら
[*20] まともな WM を目指すほどにこちらの比率が増える気がしてならない
[*21] チュートリアルと doxygen で生成された文書 (それも完全ではない)、あとはヘッダファイルだけ

オシロスコープで図形を描こう！

mernao

はじめに

みなさんオシロスコープと言うものはご存知ですね？知らなくても多分電通大に居れば 9 割の人は 1 回は基礎科学実験 A で触ることになるでしょう[*1]。今回はオシロスコープの X-Y モードを使って簡単な図形をオシロスコープに描いてみようと思います。最終的に五芒星を描くのが目標です。

ところで私は S 科です。オシロは実験でよく使います。しかしプログラミングは苦手なので、この文章には大量のクソコードが登場しますのご注意ください。時間がなかったのでオシロに表示できる事ことしか考えていません。一応作成したコードは最終章にリンクとして掲載していますので、使ってみたい人はどうぞ。

1　オシロスコープと X-Y モード

皆さんオシロスコープというと波がうねうねとしているイメージがあると思います。一番簡単なのは正弦波を入力すると数学の教科書に載ってるあの正弦波がオシロの画面に現れます。これは横軸が時間軸、縦軸が電圧になっているモードです。大抵のオシロは 2ch 以上の入力を持っているので、2 つの波形を上下に並べて見れたりします。多分実験で普通に使うのはこの時間軸モードです。今回使う X-Y モードは時間は関係ありません。縦軸に ch1 の電圧, 横軸に ch2 の電圧と言った具合に表示されます。普通に直交座標系です。例えば (ch1,ch2)=(1V,1V) となるように電圧を入力すれば対応する点が光ります。ch1 の電圧だけ変化させれば直線になります。うまい具合に ch1 と ch2 を変化させてやれば任意の一筆書きの図形が描けます。実際にオシロに表示されているのは点であり、その点が高速で移動することで残像で軌跡が描かれて図形として見えます。

そもそも X-Y モードは、その昔高精度な周波数カウンターが無かった時代に周波数を測定するために使っていたようです。2 つのチャンネルに異なる周波数の正弦波を入力すると、リサージュ曲線と呼ばれる曲線が描かれます。片方のチャンネルに厳密な周波数が分かっているセシウムなどの発振装置から信号を入力して、調べたい信号とでリサージュ曲線を作り、その形から周波数を求めていたようです[*2]。みなさんご存知の電通大の古いロゴもこのリサージュ曲線に由来するものです。この事は実際に後で確認します。

2　とりあえず音を出そう

遅くなりましたがここで開発環境を紹介します。C 言語にしたのは C 言語ぐらいしか使ったことがないからです。LinuxMint は Debian 系で、Ubuntu の派生ディストリビューションです。Gnome3 という邪悪な統合環境から簡単に逃れられるので好きです[*3]。Ubuntu の次ぐらいに人気のあるディストリビューションとか言う話を聞きました。

OS	LinuxMint14
言語	C 言語
コンピュータ	ASUS U24A
その他	アナログオシロ

さて、それでは音を出してみましょう。と言っても Linux でどうやって音を出せばいいかわかりません。

*1 頑張ってください
*2 リサージュ曲線の形は 2 つの周波数の比で決まる
*3 Debian でいいじゃんという声が...

少し調べてみると、すごく丁寧に解説している Rio's Laboratory[*4]というサイトがあったのでサクッとサンプルコードを使わせて頂きました。ありがたや。とりあえずサンプルコード sound.c をコンパイルして使おうと思ったらいきなりコケました。

```
open(): No such file or directory
```

何かが Open できないようです。ソースコードで Open しているのは 38 行目の/dev/dsp です。調べてみると確かに LinuxMint には/dev/dsp がありません。どうやら Ubuntu のいつかのバージョンの時点で削られたようです。その代わり padsp[*5]というコマンドを使うと/dev/dsp をエミュレーションできるらしいです。使い方は以下の通りです。

```
$ padsp ./sound
```

無事音が鳴った思います。一番基本的な正弦波ですね。このサンプルコードは、PCM データを生成して OSS の API を使って/dev/dsp に流し込むだけの単純なプログラムのようです。案外簡単です。詳しいことは私が書くよりも元サイトを参照してもらったほうが分かり易いと思うので書きません。

3　ステレオ出力でリサージュ曲線を描く

前章でとりあえず音が鳴りましたが、サンプルコードではモノラルでしか出力できません。リサージュ曲線を描くにはスレテオ出力が必要なのでスレテオ出力ができるようにしましょう。まず 84 行目を channel=2 に書き換えます。これでステレオで出力するように設定できました。次にステレオの PCM データを生成します。どうやらステレオの PCM データは左右左右左右... と言った具合の順序で各チャンネルのデータが格納されているようです。つまり、配列 buf[2*i] に左チャンネルを、配列 buf[2*i+1] に右チャンネルを格納すれば良い訳です。とりあえず波形データ作成部分のコードは以下のようになりました。

```
1 for ( i = 0; i < BUFSIZE / sizeof(short); i ++ ) {
2     t = ( total / (BUFSIZE / sizeof(short)) ) * i;
3     buf[2*i] = SHRT_MAX * sin( 2.0 * 3.14159 * freq_l * t );
4     buf[2*i+1] = SHRT_MAX * sin( 2.0 * 3.14159 * freq_r * t );
5 }
```

左右から freq_l,freq_r で指定された周波数の正弦波が出力されます。さて、オシロの X-Y モードで実際に見てみましょう。X-Y モードは time/div つまみを一番端まで回すと X-Y モードになったりします。機種によって様々なのでいろいろいじってみましょう。あと、横軸も縦軸も同じチャンネルで表示しようとすると、当たり前ですが傾き 1 の直線が表示されるだけです。実際に表示させたのが図 1 です、なんとなくどこかで見たような図形です。これは左右に 500Hz と 600Hz を入力しているので 5:6 のリサージュ曲線です。電通大のロゴは 5:6 リサージュ曲線をあしらったものなので一致するはず！と思っていましたが何か違いました。電通大のロゴはこの一筆書きの半周期ぐらいを描いたものらしいですが、どうにも完全には一致しな

[*4] http://homepage3.nifty.com/rio_i/lab/oss/000prologue.htm
[*5] PulseAudio OSS Wrapper

かったので、もしかするとアレンジされているのかもしれません。ですが、とりあえずリサージュ曲線が描けたので先に進めそうです。

元のサンプルコードは 5 秒しか再生できないので、if 文で Write している部分を for で回してあげましょう。5 秒毎の繋ぎ目で音が途切れますが今回はそれほど問題ないので無視します。

ここで、うまく表示されなかった場合はコンピュータのボリュームを見てみましょう。ぼやけていて非常にノイズが多い場合はボリュームが小さすぎる可能性があります。ボリュームを大きくして、電圧のレンジを大きくしましょう。逆に、鮮明だが端が途切れている場合はボリュームが大きすぎて、アンプかシステムの出力限界に達しているのでボリュームを絞りましょう。当たり前ですが電圧レンジは ch1 も ch2 も同じにしましょう。縦横比がおかしくなります。

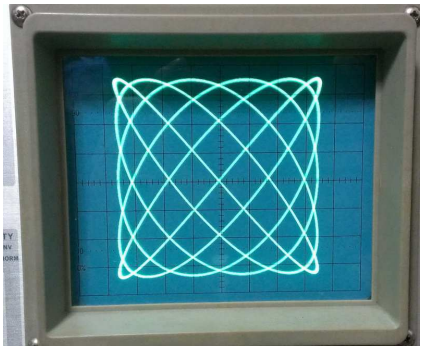


図 1　5:6 のリサージュ曲線

4　矩形波を出力する

正弦波の次は簡単な矩形波を出力します。構造は単純で、`freq_l` で指定された周波数の半周期だけ ±`SHRT_MAX` を出力します。1 周期分出力したら n=0 に戻って繰り返します。

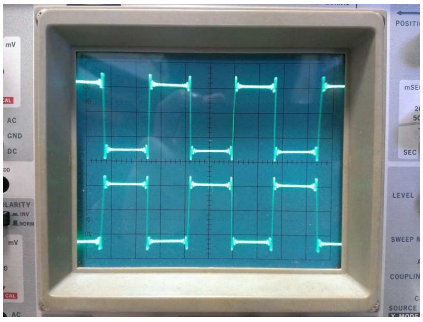


図 2　矩形波

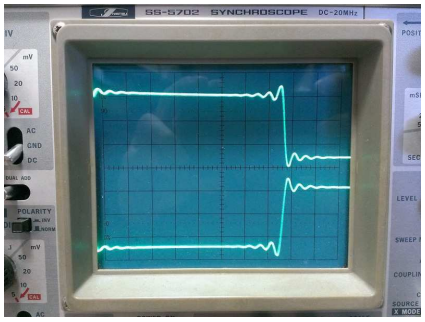


図 3　汚い波形だなぁ

```
for ( i = 0, n = 0; i < BUFSIZE / sizeof(short); i ++ , n ++ ) {
  if ( n < 44100 / ( 2 * freq_l ) ) {
    buf[2*i] = SHRT_MAX ;
    buf[2*i+1] = -SHRT_MAX;
  } else if ( 44100 / (2 * freq_l <= n ) && n < 44100 / freq_l ) {
    buf[2*i] = - SHRT_MAX;
    buf[2*i+1] = SHRT_MAX;
  } else {
    n = -1;
    i --;
  }
}
```

実際の波形は図 2 の様な感じです。時間軸を拡大してみると図 3 の様に値が変化する部分で結構歪んでます。原因はちゃんと調べていませんが、多分サウンドカードの限界とかそこら辺じゃないでしょうか。もしくはインピーダンス整合がとれていないとか。余談ですが、サウンドカードのアンプの中にはカップリングコンデンサと言うものが入っていて、直流成分を取り除いてしまうので直流の信号は取り出すことができないので注意が必要です。

5　三角波を出力する

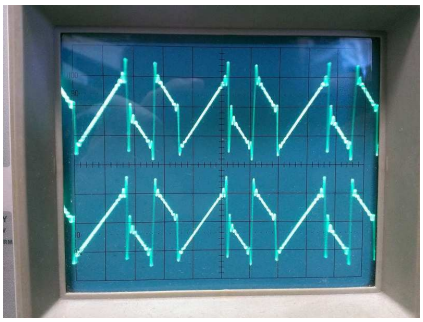


図 4　何かがおかしい

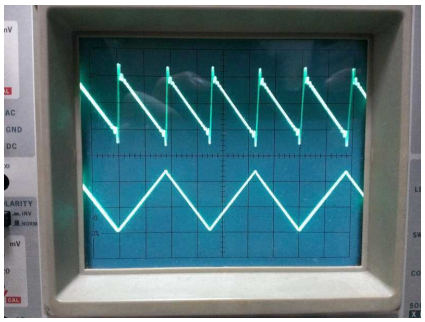


図 5　三角波と矩形波

矩形波が出力できたので今度はその次に簡単そうな三角波を出力してみましょう。値を-SHRT_MAX から +SHRT_MAX まで単調に増加させて、そこからさらに単調に減少させれば良いはずです。今はサンプリングレートが 44100Hz なので、1 サンプルでどれだけ値を増加させれば良いかを計算して変数 slope に代入しています。左チャンネルから三角波、右チャンネルからノコギリ波を出力するようなプログラムを作成して動かして見ました。しかし図 4 を見てみると明らかにおかしい事がわかります。どうやら適当に式を書いていたら符号を間違えてしまったようで、値がオーバーフローしていました。修正したものが以下のコードです。

```
slope = SHRT_MAX / ( 44100 / (4 * freq_l) );
for ( i = 0, n = 0 ; i < BUFSIZE / sizeof(short); i ++ , n ++ ) {
  if ( n < 44100 / ( 2 * freq_l ) ) {
    buf[2*i] = 0.9*(-SHRT_MAX + slope * n); // 左ch,三角波
    buf[2*i+1] = 0.9*(-SHRT_MAX + slope * n); // 右ch,のこぎり波
  } else {
    buf[2*i] = 0.9*(3 * SHRT_MAX - slope * n);
    buf[2*i+1] = 0.9*(-3*SHRT_MAX + slope * n);
  }
  if ( n > 44100 / freq_l ){
    n = 0;
    i --;
  }
}
```

配列 buf に書き込む時に 0.9 を掛けているのは、計算の誤差で括弧内の値がオーバーフローを起こす大きさでも多少は許容できるようにするためです[*6]。こいつを走らせると図 5 の様に無事三角波とノコギリ波が出力されました。

6　正方形を描いてみる

基本的な波形が生成できる様になったので、正方形を描いてみましょう。正方形を描くには図 6 のような波形を出力すれば OK です。コードは以下の通りです。

```
slope = SHRT_MAX / ( 44100 / (4 * freq_l) );
for ( i = 0, n = 0 ; i < BUFSIZE / sizeof(short); i ++ , n ++ ) {
  if ( n < 0.5 * 44100 / freq_l ) {
    buf[2*i] = 0.9*(-SHRT_MAX + slope * n); // 左ch,初め増加
    buf[2*i+1] = 0.9*SHRT_MAX; // 右ch,初めMAX
  } else if ( 0.5 * 44100 / freq_l <= n && n < 1.0 * 44100 / freq_l ) {
    buf[2*i] = 0.9*SHRT_MAX;
    buf[2*i+1] = 0.9*(3*SHRT_MAX - slope * n);
  } else if ( 1.0 * 44100 / freq_l < n && n < 1.5 * 44100 / freq_l ) {
    buf[2*i] = 0.9*(5*SHRT_MAX - slope * n);
    buf[2*i+1] = -0.9*SHRT_MAX;
  } else if ( 1.5 * 44100 / freq_l < n && n < 2.0 * 44100 / freq_l ) {
    buf[2*i] = -0.9*SHRT_MAX;
    buf[2*i+1] = 0.9*(-7 * SHRT_MAX + slope * n);
  } else {
    n = -1;
    i --;
  }
}
```

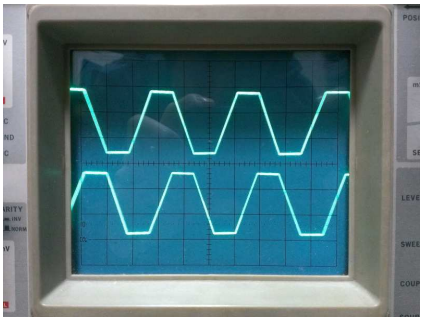


図 6　この波形を X-Y モードで見てみると…？

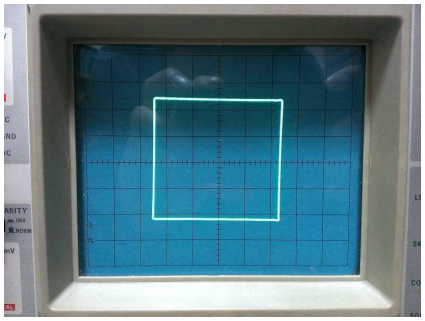


図 7　正方形

三角波と矩形波を混ぜた感じになっています。もうちょっとマシな書き方をしたかったのですが、幾分時間と知識がないので至らずという感じです。実際にオシロの X-Y モードで表示させてみると図 7 の様に見

[*6] ゴリ押しです

事に四角になっています

7　五芒星を描いてみる

いよいよ五芒星を描いてみます。五芒星というのは一筆書きで描けるいわゆる普通の星です。で、どうやって描けば良いんだ？と思うわけです[*7]。五芒星の 5 つの鋭角を全て足すと 180° です。なので 1 つの鋭角は 36° です。一筆書きで描くには 5 本の直線が必要です。

```
1  int time=100; // 1つの直線に使うサンプル数
2  for ( i = 0, n = 0 ; i < BUFSIZE / sizeof(short); i ++ , n ++ ) {
3    if ( n < time ) {
4      buf[2*i] = SHRT_MAX*sin(M_PI*18/180)*n/time; // 左ch
5      buf[2*i+1] = SHRT_MAX - SHRT_MAX*cos(M_PI*18/180)*n/time; // 右ch
6    } else if ( time <= n && n < 2*time ) {
7      buf[2*i] = buf[2*(time-1)] - SHRT_MAX*cos(M_PI*36/180)*(n-time)/time ;
8      buf[2*i+1] = buf[2*(time-1)+1] + SHRT_MAX*sin(M_PI*36/180)*(n-time)/time ;
9    } else if ( 2*time <= n && n < 3*time ) {
10     buf[2*i] = buf[2*(2*time-1)] + SHRT_MAX*sin(M_PI*90/180)*(n-2*time)/time ;
11     buf[2*i+1] = buf[2*(2*time-1)+1] - SHRT_MAX*cos(M_PI*90/180)*(n-2*time)/
           time ;
12   } else if ( 3*time <= n && n < 4*time ) {
13     buf[2*i] = buf[2*(3*time-1)] - SHRT_MAX*cos(M_PI*36/180)*(n-3*time)/time ;
14     buf[2*i+1] = buf[2*(3*time-1)+1] - SHRT_MAX*sin(M_PI*36/180)*(n-3*time)/
           time ;
15   } else if ( 4*time <= n && n < 5*time ) {
16     buf[2*i] = buf[2*(4*time-1)] + SHRT_MAX*sin(M_PI*18/180)*(n-4*time)/time ;
17     buf[2*i+1] = buf[2*(4*time-1)+1] + SHRT_MAX*cos(M_PI*18/180)*(n-4*time)/
           time ;
18   } else { // 最後まで行ったらnをリセットして最初に戻る
19     n = -1;
20     i --;
21   }
22 }
```

1 行目で、星の 1 辺に使うサンプル数を設定します。100 なので 1 辺を描くのに 100/44100 秒かかります。なので星を描くのには 500/44100 秒かかります。今回はややこしいので `freq` で周波数を指定しません。このプログラムの描画の開始点は☆[*8]の頂点で、そこから右側に描くことを考えています。とりあえずオーバーフローしないように適当にマージンをとってあります。作ってから気がついたのですが、多分左チャンネルの値が負になる事は有りません。しかし、とりあえずカップリングコンデンサで直流成分は取り除かれてしまうので星はオシロの中央に表示されるので良いことにします。n=100 に達すると次の if 文に入り、次の辺が描画されます。n=100〜199 の時は n=99 の時の電圧を引き継いでそこに足すようにしてあります。同じ事を繰り返すと最後には元の点に戻ってくるはずです。`SHRT_MAX*sin(M_PI*18/180)` は、n=99 の時

[*7] 自分の図形問題を解く能力があまりにも退化していて驚きました
[*8] 実際は中抜きではない

の左チャンネルの値となっています。これを 0~1 まで 0.01 ずつ増加させています。まぁつまり sin と cos で角度を増分 (角度) を指定している訳です。

わかりにくい説明をしていても仕方ないので、X-Y モードで見てみましょう。はい！五芒星が表示できました！上下逆さまになってしまいましたが五芒星です。矩形波の時は汚くて心配しましたが、意外と綺麗に表示できました。とりあえず目標達成です。

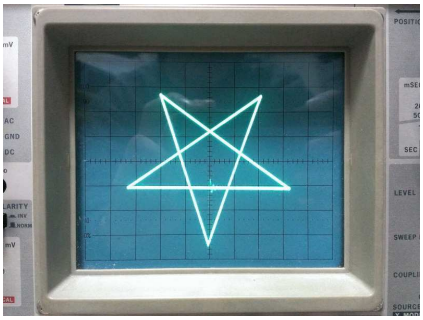


図 8　五芒星！

図 9　実験中の様子

8　おわりに

図 9 は実験中の写真です。私は先進理工学科電子工学コースなので、こういう風景はいかにも電子工学科という感じがして好きです。

実は、自分で作りたいものをプログラミング言語を使って作るのはこれが初めてです。手探りでゴリ押しで作ったので各方面からツッコミが来まくりそうなクソコードですが、なんとか完成してよかったです。実際は作業時間は短いのですが。今度はもうちょっと工夫して、図形が動くようになったりしたらいいなとか思っています。あと、クソコードをなんとかしたい。

9　作成したコード一覧

作成したコードへのリンクです。オシロに星を表示したいぜ！という人はクソコードですが使ってみてください。リサージュ曲線を表示させるだけなら、Windows に WaveGene という波形発生ソフトがあるので使ってみてください。というか任意の波形を出力できるらしいのでぶっちゃけこれで星も描けそうです。え？オシロスコープが無い？MMA の部室に 2 台ありますよ！

ソースコード置き場: `http://wiki.mma.club.uec.ac.jp/mernao/hyakuman2013spring`

参考文献

[1] Linux Sound programming with OSS API
`http://homepage3.nifty.com/rio_i/lab/oss/000prologue.htm`

掃き出そう

zico

はじめに

実数成分から成る行列に行基本変形を施し、階段行列に簡約化する Ruby スクリプトを書いた。

1 階段行列

1.1 行基本変形

(R1) ある行を c 倍する ($c \neq 0$)

(R2) ある行を何倍かして別の行に足す

(R3) 2 つの行を入れ替える

1.2 階段行列

定義

行列 $A = \begin{bmatrix} \vec{a_1} \\ \vec{a_2} \\ \vdots \\ \vec{a_m} \end{bmatrix}$ が以下の条件を満たすとき、A を階段行列という。

(i) $\vec{0}$ である行は下に集まっている

(ii) $\vec{a_i} \neq \vec{0}$ なら 1 番左の 0 でない成分 (主成分:main component) は 1

(iii) 主成分は下にいくほど右にある

(iv) 主成分を含む列の主成分以外の成分は全て 0

定理

- 任意の行列は何回かの行基本変形を施すことにより階段行列に簡約化できる
- 階段行列はもとの行列から一意に決まる

1.3 行列の簡約化:掃き出し法 (sweep-out method)

Step1 全てが 0 でない列のうち、1 番左の列の 1 番上の成分が 0 なら行を 0 でないものと入れ替える (R3)

Step2 第 1 行を何倍かして、主成分を 1 にする (R1)

Step3 主成分にある行を何倍かして他の行に足し、主成分の上下の成分を 0 にする (R2)

Step4 第 1 行を除いた行列について、Step1 から Step3 を繰り返す

2　掃き出し法による逆行列の導出

A が逆行列を持つとき、$AX = I \iff XA = I$. よって $AX = I$ となる X が A の逆行列である。
$B = [\vec{b_1}, \vec{b_2}, \cdots, \vec{b_n}], I = [\vec{e_1}, \vec{e_2}, \cdots, \vec{e_n}]$

$$\begin{cases} A\vec{b_1} = \vec{e_1} \\ A\vec{b_2} = \vec{e_2} \\ \quad\vdots \\ A\vec{b_n} = \vec{e_n} \end{cases}$$

を解けば、すなわち $[A|I]$ を簡約化すれば、$[\vec{b_1}, \vec{b_2}, \cdots, \vec{b_n}]$ すなわち B が求まる。
つまり $[A|I] \to$ (行基本変形) $\to [I|B]$

3　書いたスクリプト

実数値から成る配列の配列を受け取り、各要素を Ratinoal(有理数) オブジェクトとして計算する。計算結果を確かめるため、逆行列を掃き出し法で求め、もとの行列に掛けた。Ruby のバージョンは 1.9.3 である。なお、行列の乗算にはライブラリ matrix を用いた。Matrix クラスでは共役複素数を扱ったり、逆行列、固有値・固有ベクトルを求めたりできるが、行列の変形にはあまり有用でない。

ソースコード 1　sweep-out.rb

```
def sweep_out(matrix, main_i=0, main_j=0)
  # Step1(R3)
  matrix[main_i..-1] = matrix[main_i..-1].sort_by do |r|
    r.map!{|e| e.to_r}
    r.index(r.find{|e| e!=0}) || r.size
  end
  while b = matrix[main_i] && matrix[main_i][main_j] do
    if b==0 # Not main component
      main_j+=1
      next
    else
      matrix.each_with_index do |r, i|
        main = matrix[main_i][main_j]
        if i==main_i then
          # Step2(R1)
          r.map!{|e| e / main}
        elsif r[main_j]!=0 then
          left = r[main_j]
          # Step3(R2)
          r.each_with_index do |e, j|
            r[j] = e - matrix[main_i][j] / main * left
          end
        end
      end
      # Step4
      sweep_out(matrix, main_i+1, main_j)
      return matrix
    end
  end
end

# print matrix
def printm(ary)
  ary.each{|r| puts r.join("␣")}
  puts
end
```

```ruby

# example
a = [[1,3,6,7,4],
     [3,6,8,6,3],
     [4,6,8,7,7],
     [2,3,4,4,2],
     [7,4,4,3,9]]

ai=[]
a.each_with_index do |r, ri|
  ai << r + Array.new(a.size){|ii| ri==ii ? 1 : 0}
end
puts "[A|I]="
printm ai

b=[]
ib = sweep_out(ai)
ib.each do |r|
  b << r[a.size..-1]
end
puts "[I|B]="
printm ib

require "matrix"

a, b = Matrix[*a], Matrix[*b]
puts "A*B="
printm (a*b).to_a
```

実行結果

```
[A|I]=
1 3 6 7 4 1 0 0 0 0
3 6 8 6 3 0 1 0 0 0
4 6 8 7 7 0 0 1 0 0
2 3 4 4 2 0 0 0 1 0
7 4 4 3 9 0 0 0 0 1

[I|B]=
1/1 0/1 0/1 0/1 0/1 14/57 20/57 -47/57 4/19 7/19
0/1 1/1 0/1 0/1 0/1 -62/57 -56/57 86/57 23/19 -12/19
0/1 0/1 1/1 0/1 0/1 31/38 47/38 -43/38 -63/38 9/19
0/1 0/1 0/1 1/1 0/1 -2/19 -11/19 4/19 20/19 -3/19
0/1 0/1 0/1 0/1 1/1 -2/57 -11/57 23/57 -6/19 -1/19

A*B=
1/1 0/1 0/1 0/1 0/1
0/1 1/1 0/1 0/1 0/1
0/1 0/1 1/1 0/1 0/1
0/1 0/1 0/1 1/1 0/1
0/1 0/1 0/1 0/1 1/1
```

掃き出し法により実行列を簡約化できた。

Twitter のリストを自動生成してみる

早川　敦士 (hayakawa)

はじめに

twitter でフォローしている人が増えてくると、TL の流速が早くて読むのが大変になってくるので、リスト管理をすると、便利になる。しかしながら、リストの作成をクラスタ毎に手動で作っていくと、かなりの労力を必要とする。なので、これを自動化してみた。

1　方法

1.1　データ

twitter で、自分がフォローしているユーザにおいて、それぞれのユーザ同士でフォローしているかどうかのデータを表 1 のように取得する。

表 1　データのイメージ

	hoge	piyo	foo	bar	spam
hoge	0	1	0	1	0
piyo	1	0	0	0	1
foo	0	0	0	1	1
bar	1	0	0	0	1
spam	0	0	0	1	0

1.2　手法

階層的クラスタリングであるウォード法 [1] を利用した。階層的クラスタリングとは、個体間の類似度（ユークリッド距離など）が近いものを順番に集めてクラスタを作っていく方法であり、樹形図で表現される。クラスタを作る時は、木の高さが一定になるようにする

また、文献 [1] にウォード法について、次のように記されている。

> ウォード法 (Ward's method) は、２つのクラスターを融合した際に、群内の分散と群間の分散の比を最大化する基準でクラスターを形成していく方法である。ウォード法は最小分散法 (minimum variance method) とも呼ばれている。

クラスタ重複を考慮するために、文献 [2] を参考にし、樹形図の高さを 2 種類選択した。

2　結果

樹形図は、図 1 のようになった。一番上の部分から見ていき、それぞれのクラスタがどのように別れているかを見ることを出来る。

今回は、クラスタが 3 つに分かれる部分と 5 つに分かれる部分に注目した。それぞれ、UEC クラスタ、データマイニングクラスタ、その他のユーザー群といった状況になっている。また、さらに掘り下げると、UEC クラスタの中で、いつも TL にいる人・時々TL にいる人と別れた。また、データマイニングクラスタは 2 つに別れて、影響力の強うそうな人たちでクラスタが構成されているのが現れた。

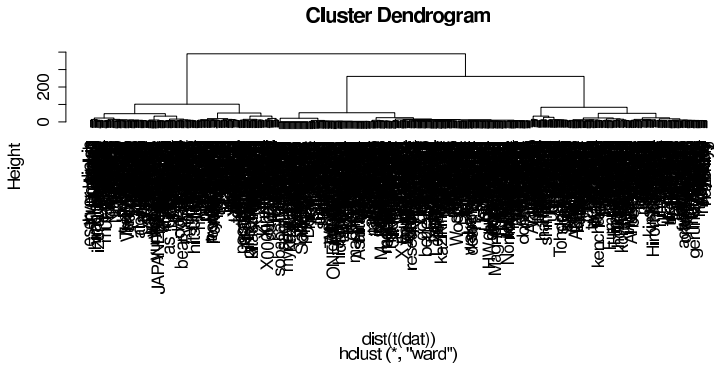


図 1　クラスタリングの結果

3　考察

クラスタ毎に解釈を行ってみたが、必ずしも綺麗に分かれてはいなかった。UEC クラスタの人がその他に分類されたり、データマイニングクラスタでない人がここに入ってきたりしていた。今回は、距離としてユークリッド距離を利用したが、データがスパースになっている（0 がたくさんある）場合は、コサイン類似度を利用した方が良いのかもしれない。
また、フォローフォロワーの関係は、ネットワーク構造になっているので、これを活かしたクラスタリングの方が適切かと感じた。

4　今後

今回は、フォローしているか否かでクラスタリングを行ったが、フォローされているかどうか、既に作成されたリスト、プロフィール情報等といったデータの活用も試してみたい。このようなリスト作成機能を web サービス化出来たら、面白いかもしれない。

参考文献

[1] R とクラスター (1),http://www1.doshisha.ac.jp/ mjin/R/28/28.html,2013 年 3 月 18 日 アクセス
[2] 小沢 一雅 (2002) 分類における重複性の表現手法： 重複クラスタリング,IPSJ SIG Notes 2002(52), pp1-8, 2002-05-31

百萬石 2013 年 春号 ©電気通信大学 MMA

2013 年 4 月 3 日　初版第一刷発行　【本書の無断転載を禁ず】

著　者　alstamber, clear, hayakawa, hirosun
　　k_operafan, mernao, mznh, zico
編集者　日野 樫雄 (hiyakashi)
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
　　〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
　　http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室
表　紙　電気通信大学 MMA

MMA

こんにちは. 電気通信大学 MMA です。

MMA は
, 1975年に創設
された, 大学公認の文化
系サークルです.

設立当初はマイコンの設計や製作
を主に行っていたそうですが,
現在はソフトウェアやネットワーク
まわりの管理・研究・開発などを主
に, 計算機に関わることだったら
なんでもやってみようというスタン
スで活動しています.

http://wiki.mma.club.uec.ac.jp

印刷 / 電気通信大学 MMA
Printed in Japan

hyakumangoku
Spring 2013
発行人 /MMA　編集 /hiyakashi　２０１３年４月１日発行
発行 / 電気通信大学 MMA　〒 182-8585　東京都調布市調布ヶ丘 1-5-1　サークル会館 2 階
表紙デザイン / hiro1357　表紙写真 /hiro1357
TAKEFREE