目次

ゲージの作り方

ピーポー

1　はじめに

今回は、DX ライブラリを使ってゲージを作る方法を記載します。この記事を作成した理由は、自分がゲージを作ろうと思ったときに、押しっぱなしに対応したものが作れなくて苦戦したので、そういったことで悩んでいる人のためになると思ったからです。

2　本文

DX ライブラリで素材なしで作れるゲージのプログラムです。必要に応じて、ゲージのふちなどの素材を用意してください。また、枠の「DrawBox」は FALSE にしておかないと、中身が塗られた状態になってしまいます。

3　主なソースコード

```
1  if (sf1 == 0)
2  {
3      gaugecounter = gaugecounter + 1;
4  }
5
6  if (sf1 == 1)
7  {
```

```
8       gaugecounter = gaugecounter - 1;
9   }
10
11  if (gaugecounter >= 120 && sf1 != 2 )
12  {
13      sf1 = 1;
14  }
15
16  if (gaugecounter <= 0 && sf1 != 2 )
17  {
18      sf1 = 0;
19  }
20
21  if (timecounter >= 199999999
22   && CheckHitKey(KEY_INPUT_SPACE) != 1)
23  {
24      if (gaugecounter < 120)
25      {
26          sf1 = 0;
27      }
28
29      if (gaugecounter >= 120)
30      {
31          sf1 = 1;
32      }
33
34  }
35
36  if (CheckHitKey(KEY_INPUT_SPACE) == 1)
37  {
38      upflag++;
39      sf12 = 1;
40  }
41
42  else
43  {
44      upflag = 0;
45  }
46
47  if (upflag == 1)
48  {
49      sf1 = 2;
50  }
```

```
51
52 if (sf12 == 1)
53 {
54     timecounter = timecounter + 1;
55     if (timecounter == 200)
56     {
57         sf12 = 0;
58         timecounter = 0;
59         sf1 = 1
60     }
61 }
```

上記のボックスの中身がゲージに関する主なコードになっています。「sf1」でフラグの管理を行っています。「DrawBox」の中身をこの文字に対応させれば、時間に応じてゲージの増減を表現できます。また、枠に当たる部分に素材を配置すれば、もっと良いゲージになると思います。キーボード等の押しっぱなしにも対応できるようになっています。「upflag」の中身が、1の瞬間のみ、キーボードからの入力を受け付けるようにしています。「sf1」が2になった時点で、ゲージを停止するようにしています。また、「timecounter」の値が２００を越えた時点で再度ゲージが動くようになっています。そのときのゲージのパーセンテージを保存したい場合は、配列を作って「if (upflag == 1)」の中に設置してください。ゲージを一度で使いきりたい場合は、最後の「sf1」を削除してください。

4 おわりに

今回は学園祭に向けて作成したゲームのソースコード中で一番苦戦した部分をこちらに記載させていただきました。このコードが少しでも参考になれば幸いです。

Twitter: @piipoo127

2016年7月13日

2 次元ゲーム画面における弾いた球の動き

柴吉

1　はじめに

私は今年の 6 月に開催された学園祭にてクリックにてボールを弾き、既定の条件を満たすまでに何度ボールを弾くことができるかを試すミニゲームを作成した。

その際に使用したクリック時のボールの挙動の変化と使用するか悩んだ結果使わなかった挙動の変化について述べる。

2　クリック後のボールの挙動

2.1　速さ

今回作成したゲームではクリックするごとに速度を上げていく要素を取り入れたため逐次値を上げていく変数 $speed$ とボールの中心座標 (x, y) とクリックした座標 (xx, yy) を用いて速度 (vx, vy) を以下のように表した。

$$vx = ((x - xx)/(x - xx + y - yy)) * speed \tag{1}$$
$$vy = ((y - yy)/(x - xx + y - yy)) * speed \tag{2}$$

この数式を使用した理由としては移動した距離 $(vx + vy)$ が $speed$ と同値になることで $speed$ の増加を鮮明にわかるようにするためといったことが挙げられる。

2.2　あたり判定

この方式を用いると $x-xx$ と $y-yy$ をともに使用するため、ついでに $(x-xx)^2+(y-yy)^2$ も算出しておく。

するとこの値とボールの半径を比較することによってクリックした地点がボールの範囲内であるかを判定することができる。

2.3　使用を断念した動きの補正

今回は $speed$ の上昇を感じてもらうために断念した方法としてボールの中心から離れた位置をクリックするほど速く動くようにするというものである。

方法とはいえど内容はシンプルであたり判定のときに求めた $(x-xx)^2+(y-yy)^2$ を vx と vy にそれぞれ乗算しただけである。

これにより初めはこの要素を追加してただ弾くのではなく動きも制御できるようにすることでゲーム性が高まると考えてのことであったのだが、作成していく過程で $speed$ をある程度上げてしまうといくら頑張ろうともボールの中心を狙うのは困難になることが判明したのでこの方法をやめて純粋にクリックするだけのゲームとした。

3　今回この題材を選択した理由

これまで何度か学園祭などで展示するためのゲームを作ってきて感じたこととして、複雑な動きを作るうえでもボールの動かし方やあたり判定といった基本的なことの組み合わせや応用だったりすることばかりだということがある。

今回述べたこの手法が効率などに適しているとは言えないがゲームを作るうえで参考にすることで改良方法を思いつく可能性もあると私は考えている。

したがって今回このテーマにした意味はこれを読んで「自分ならこう応用する」、「ここはダメだろう」といったことを考えてもらうことで基本事項の充実を図れるのではないかと考えたからである。

4 おわりに

あたり判定の効率化はもちろん、動きをリアルっぽく見せたいなどゲームを作るうえで求めていきたい項目は数多く存在する。

今後プログラミングをしていくなかでその項目を満たしていきたいと考えているが、まずは基本の充実が大事だなとも考えている。

そのためにもこういった基本的な、「そんなことか」と思ってしまいそうなことでも他人のやり方を調べたり自分なりに考えたりすることでよりよいプログラムをかけるように努力したいと思う。

2016 年 7 月 20 日

TEX 使用のレポートに使えるコマンド集

腹痛

1　前置き

　私はよく大学のレポート等を TEX で作っています。
うまく使えれば、すごく綺麗で見栄えの良いレポートを作成することが出来ます。
しかし、私は TEX において初心者であるので、レポートを作ってる際に「ここをこうしたいのにどうすればこう出来るのか分からない！」と思うことが多いのです。

　そこで私は初心者なりに勉強して、初心者の頃に知っておくと便利だと思ったことを書いていこうと思います。
項目としては、オススメのエディタ・図の位置について・小テクニックの 3 つに分かれます。

2　オススメのエディタ

　まず、エディタの選択です。既にお気に入りのエディタがある人はここの項目はあくまで参考にすると良いでしょう。
Windows ではオフラインなら TeX インストーラー 3(インストーラー)、オンラインなら CloudLaTeX(サインインが必要) です。

Linux は元々入っているものや、CloudLaTeX を使うとよいでしょう。
Mac は私が使っていないので CloudLaTeX 以外はオススメすることができません。

3 図の位置について

前は図を置くときに自分の置きたい場所に置くことが出来ませんでした。
しかし、いくつかの方法を用いることによりある程度はコントロールできるようになりましたのでそれらを紹介します。

1. 位置指定を使う
   位置指定とは、

   `\begin{figure}[ここの括弧]` ←

   の部分の記述です。種類には、p(新しいページに置く), h(出来るだけ記述されている場所に置く), t(ページ上部に置く), b(ページ下部に置く)、の 4 つがあります。
   私は出来るだけ文章の近くに図を置きたいことが多いので h をよく使っています。また、エクスクラメーションマーク (!) を!h のように使うと、普通よりそこに置く優先度が上がり置きやすくなります。普通の状態でうまくいかないときは使ってみてください。
2. `\hspace*{(x)cm}`を使う
   これを使うと物理的に図を上へ移動させることが出来ます。
   記述例

```
1 \begin{figure}[h]
2 \vspace*{-3.4cm}%図を上へ３．４cm移動
3 \includegraphics[scale=0.8]{test.pdf}
4 \caption{test}
5 \label{fig:test}
6 \end{figure}
```

3. wrapfig を使う
   wrapfigure を使うと図の周りを沿うように文が配置されます。小さい図を貼って横に説明等を書く場合にオススメです。記述例

```
\begin{wrapfigure}[5]{r}[3mm]{30mm}
\includegraphics[scale=0.8]{test.pdf}
\caption{test}
\end{wrapfigure}
```

   パラメータは自分で実験的に調節してください。

4 小テクニック

ここには少し役立つテクニックと呼ぶかどうか微妙なレベルのテクニックを書いていきます。

1. \section{}や\part{}に番号を付けない
   の前にアスタリスク (*) を付ければよい。

2. listing を使った際に半角文字が右に寄ってしまう場合
   jlisting を使うか、半角文字をすべて全角文字に変える。

3. ページ下部に現在のページ番号/ページ合計を表示したい
   パッケージの fancyhdr と lastpage を使えばよい。
   記述例

```
\usepackage{fancyhdr}
\usepackage{lastpage}

\fancypagestyle{test}{%ページのスタイルを定義する
\cfoot{\thepage/\pageref{LastPage}}
}
\pagestyle{test}

\begin{document}
```

```
10 ---something-----
11 \end{document}
```

4. 上部に出る横線を消したい
   3 の記述例の 5 行目と 6 行目の間に

   ```
   \renewcommand{\headrulewidth}{0.0pt}
   ```

   を追加する。
5. 図に pdf や png を使いたい
   パッケージの graphicx を追加する。とりあえず、

   ```
   \usepackage[dvipdfmx]{graphicx}
   ```

   と書いておけば pdf 形式や png 形式も図に使うことが出来る。

5 おわりに

使えそうな技術はありましたでしょうか。あくまで初心者向けなので、出来る人たちからしたら知っているものばっかりだったかもしれません。
しかし、少しでも既知でない知識を与えることが出来たなら幸いです。
自分で TEX を書きながら試行錯誤でより便利に、また綺麗にレポートを書けるようになっていきましょう。

2016 年 7 月 20 日

Boost.Spirit.Qi を使用して PNG ファイルのパーサを作る

りぶ

1　はじめに

独自に作ったコンピュータ言語[*1]をプログラムで扱うとき、Yacc などのパーサジェネレータや Boost.Spirit.Qi などのパーサコンビネータを使用します。これらは通常、テキストを解析をしてくれるものを生成するのですが、Boost.Spirit.Qi はなんとバイナリファイルのパーサも書けちゃいます。

書けるパーサは先頭からズラッとチャンクが並んでいるようなバイナリファイルで、ファイルのオフセットが書いてあるようなものは (多分) できないです。できたら僕に教えてください。

2　Boost.Spirit.Qi とは

Boost.Spirit.Qi(以下 Qi) は、C++ 用のパーサコンビネータライブラリです。小さい部品 (パーサ) を組み合わせて構文解析器を作ることができます。

Qi の使い方に関してはインターネット上の各記事や、僕のブログ[*2]で解説記事を

[*1] 機械語ではなくプログラミング言語やデータ記述言語などの形式言語のこと。

[*2] 現在連載中。 http://sparelibs.hateblo.jp/

公開しているので、そちらを参照してください。

3 PNG ファイルの構造

今回は画像フォーマットの 1 つである PNG[*3]ファイルをパースしてみようと思います。

PNG ファイルの先頭 8 バイトは `89 50 4E 47 0D 0A 1A 0A` と決まっています[*4]。文字列リテラルで表すと`"\x89PNG\r\n\x1a\n"`となります。その次には、PNG のチャンクが並んでいます。

表 1 PNG チャンクフォーマット

サイズ	名前	説明
4 bytes	チャンクデータのサイズ n	ビッグエンディアン
4 bytes	チャンクタイプ	ASCII 文字 4 文字で表す
n bytes	チャンクデータ	
4 bytes	CRC-32	誤り検出符号

このチャンクフォーマットに従ったチャンクが隙間なく並んでいます。

4 パーサを書いてみる

さて、PNG ファイルのざっくりとした構造が分かったので、Qi を使ってパーサを組んでみましょう。

```
1 // 標準ライブラリのインクルードは省略
2 #include <boost/spirit/include/qi.hpp>
3 #include <boost/phoenix.hpp>
4
5 namespace qi = boost::spirit::qi;
6 namespace phx = boost::phoenix;
```

[*3] Portable Network Graphics

[*4] いわゆるマジックナンバー。PNG であることをコンピュータが識別できる。

```
7
8  struct png_chunk {
9      uint32_t size;
10     std::string type;
11     std::string data;
12     uint32_t crc;
13 };
14 BOOST_FUSION_ADAPT_STRUCT(png_chunk, size, type, data, crc)
15
16 template <typename Iterator>
17 struct png_parser
18     : public qi::grammar<Iterator, std::vector<png_chunk>()> {
19
20     png_parser() : png_parser::base_type(expr) {
21         expr %= "\x89PNG\r\n\x1a\n" >> *chunk;
22         chunk %= qi::big_dword
23             >> qi::repeat(4)[qi::char_]
24             >> data(phx::at_c<0>(qi::_val))
25             >> qi::big_dword;
26         data %= qi::repeat(qi::_r1)[qi::byte_];
27     }
28
29     qi::rule<Iterator, std::vector<png_chunk>()> expr;
30     qi::rule<Iterator, png_chunk()> chunk;
31     qi::rule<Iterator, std::string(uint32_t)> data;
32 };
```

まず、表 1 に従って png_chunk 構造体を定義します。チャンクデータはバイト配列なので、std::string で表せます。元の PNG ファイルのバイト配列から std::vector<png_chunk>へ落とし込む作戦です。

BOOST_FUSION_ADAPT_STRUCT マクロで、png_chunk 構造体を Qi で使えるように仕組みます。

さて、今回のメインである png_parser 構造体ですが、コンストラクタ内に文法が定義してあります。文法 expr をルートとしています。expr は、PNG の最初の 8 バイトの文字列の後にチャンクが複数続くことが表されています。

文法 chunk の説明は後に回して、まずは文法 data について解説します。data は uint32_t の値を引数としてとり、その値の長さ分のバイト配列をパースする文法になっています。qi::repeat ディレクティブで qi::byte_を繰り返す回数を制御し

ています。引数は qi::_r1 で受け取れるので、それを使用すれば渡された長さのバイト配列を取ることができます。

文法 chunk では、PNG チャンクの構造を表しています。まず、qi::big_dword で文字列をバイナリとみなして、ビッグエンディアンの 4 バイト符号なし整数を取ります。次に、先ほどと同じように qi::repeat ディレクティブを使用して、4 文字のチャンクタイプをパースします。次に、先ほどの文法 data を使用することになりますが、引数に phx::at_c<0>(qi::_val) とあります。これは Boost.Phoenix を使用して、結果が格納されている変数 qi::_val の 0 番目 (最初) の要素を phx::at_c<0>() で取得しています。つまり、最初に qi::big_dword で取得したチャンクサイズの値を文法 data で使用することができます。一番最後に CRC の値を qi::big_dword で取得して文法は完成です。

5 実際にパースしてみる

ファイルを読んで、qi::parse() 関数でパースすれば終わりです。

```
1  int main() {
2      auto path = /* ファイルパス */;
3      std::ifstream ifs(path, std::ios::in | std::ios::binary);
4      std::string buf((std::istreambuf_iterator<char>(ifs)),
5          std::istreambuf_iterator<char>());
6      ifs.close();
7
8      png_parser<decltype(buf.begin())> parser;
9      std::vector<png_chunk> result;
10     qi::parse(buf.begin(), buf.end(), parser, result);
11
12     for (auto&& a : result)
13         std::cout << a.type << ":" << a.size << std::endl;
14
15     return 0;
16 }
```

これを実行すると、チャンクタイプとチャンクサイズが標準出力に出力されます。一旦パーサが書けてしまえばとても簡単にパースできるようになるのは Qi の便利なところです (パーサを書くのが少し難しいですが)。

6　おわりに

本来はコンピュータ言語のパーサを作る Qi で、バイナリファイルのパーサを作ってみました。Qi が含まれる Boost.Spirit は応用範囲がとても広いライブラリです。みなさんもぜひ使ってみてください。今回紹介した PNG パーサを改造して、さらに詳細にパースをしたりしてみるのも面白いかもしれません。

Twitter: @g_s_sequence
GitHub: gssequence
Blog: http://sparelibs.hateblo.jp/
2016 年 7 月 19 日

四分木空間分割

ウバタマ

空間を 4 分割していき決まった法則で番号をふる。これをモートン序列と呼ぶ。

ルート空間を 4 分割し、親空間を作る。その時図のように番号を振る。次に親空間のそれぞれを 4 分割し、子空間を作り同じように番号を振る。同様に孫空間も作る。

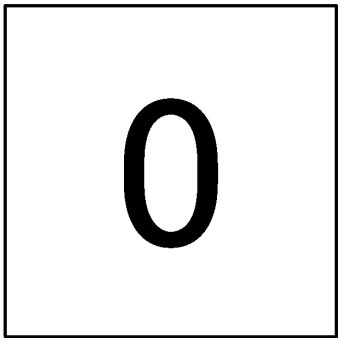


図 1　ルート空間

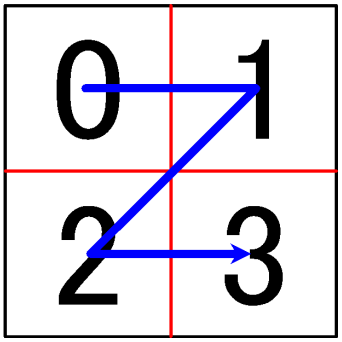


図 2　親空間

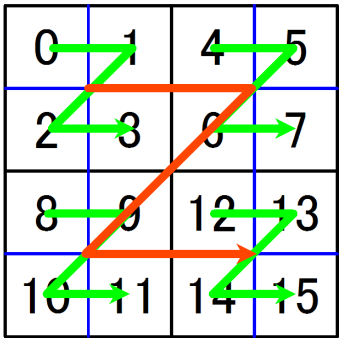


図 3　子空間

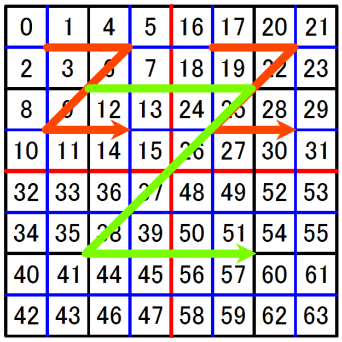


図 4　孫空間

モートン序列に並んだ空間は上位空間の情報を持つ。例えば、孫空間の 45 番の場所は親空間から順に 2 番、3 番、1 番となる。ここで 45 を 2 進数に変換すると、101101 となる。それぞれ 2bit ずつ見ると上位空間の順番になっている。

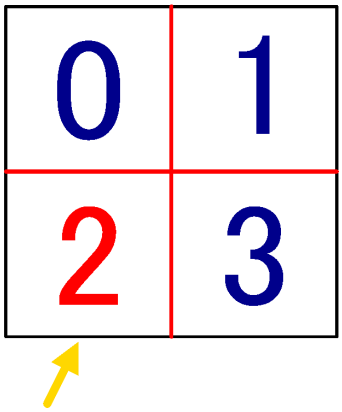


図 5　親空間

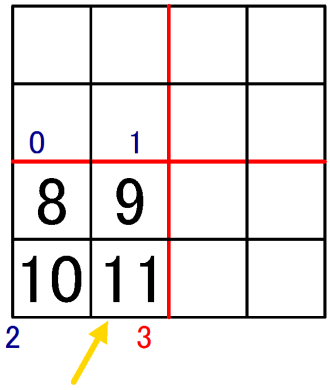


図 6　子空間

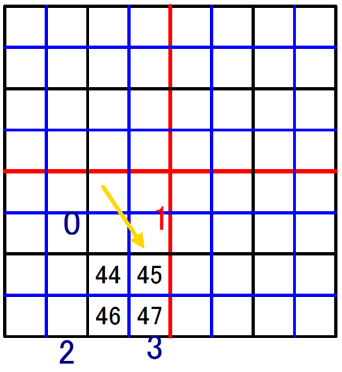
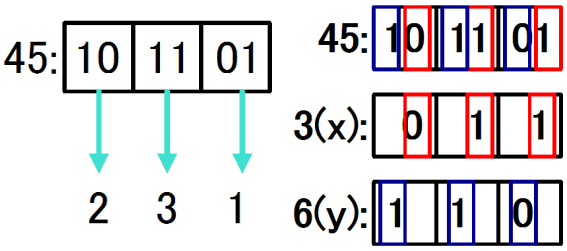
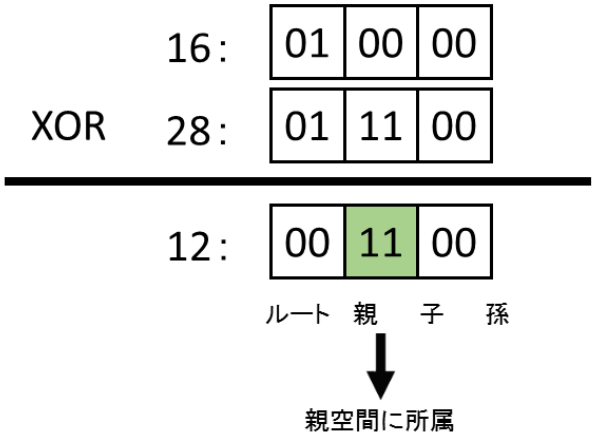


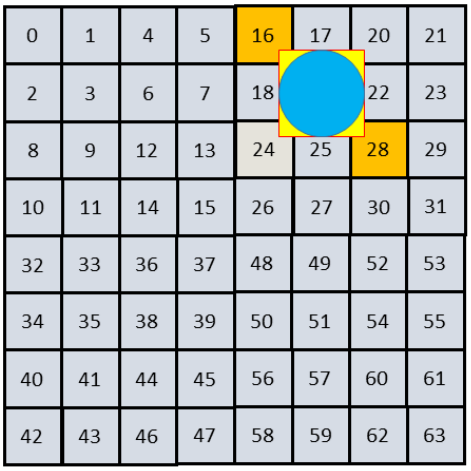
図 7　孫空間



またモートン序列はマスの数の x 座標 y 座標の情報も持っており、2bit ずつ区切った 1bit 目が x 座標、2bit 目が y 座標である。

点の所属空間が分かったので、これを用いてオブジェクトの所属空間を出す。





オブジェクトの左上と右下の点が所属する番号で排他的論理和をとる。こうしてでた値を 2 進数に変換し 2bit ずつ区切る。0 でない値となっている区画がこのオブジェクトの所属空間となる。

2016 年 7 月 22 日

OpenCV でお手軽擬似的クロマキー合成

たち

1　はじめに

皆さんこんにちは。今回は OpenCV を利用した簡単なクロマキー合成について書きました。発展的な内容は無く、これから OpenCV を学びたい人向けな内容となっています

2　クロマキー合成とは

クロマキー合成を説明する前にまずマスク処理について説明します。マスク処理というのは入力画像をマスク画像によって指定した範囲のみ出力できるように処理する技術のことである。・・・・・・と言っても分からない人にはなんのこっちゃでしょう。誰でも分かるようにクッキー作りにたとえて説明すると、こねた後のクッキーの生地の塊が入力画像で、マスク画像とはクッキーの型です。ここでは星形としましょう。その二つを使ってできる星形のクッキー生地が出力画像というわけです。コラ画像等を作ったりするのにも使える技術ですね。次はクロマキー合成についてです。クロマキー合成というのは任意の色を透過色として設定してマスク処理と同じような処理をし、複数画像を合成する処理のことです。例えば天気予報の映像にもこの技術は使われています。入力画像がスタジオのカメラで撮影している実際の映

像で、映像が映される部分は緑や青色の壁になっています。その色を透過色として設定することでそこに天気予報の映像を重ねることができます。この記事では入力画像をカメラからの映像とし、マスク画像を動的に生成してマスク処理を行う手法を紹介します。

3 処理の概要

さて、今回はタイトルにある通りお手軽『擬似的』クロマキー合成を紹介します。流れを説明しますと、まずカメラから入力画像を取り入れます。それをグレースケールに変換し、ガウシアンフィルタで濃淡画像にします。それを二値化することで黒と白だけの画像にします。この白い部分を透過部分としてマスク画像として使う。マスク画像と入力画像の論理積をとり、切り出し画像 1 とします。これが背景の上に張り付ける画像です。次にマスク画像の否定をとり、マスク画像を反転させます。反転させたということは、切り出した画像の部分以外を切り出すためのマスク画像ということになります。反転マスク画像と背景画像の論理積をとり、切り出し画像 2 とします。切り出し画像 1 と切り出し画像 2 はお互いに自分たちの画像に無い部分を補完しあっていることになりますね。なのでこの二つの画像の論理和をとると、背景画像の上に透過処理を施した画像を張り付けたように見える画像が出来上がります。これを while 文で常に動かせば動的にマスク画像を生成し結果を出力することができます。基本的な部分はこれで終わりですが、次の問題が透過色の設定です。このプログラムは『擬似的』クロマキー合成ですので、透過色を設定することはできません。ではどうやって透過させるのか？それはマスク画像を作りだす部分が肝なのです。マスク画像を作る際に二値化を行っていますよね。これはグレースケールにした後濃淡画像にしたものの画素を閾値を境に 0 と 1 に分けるという処理なのですが、その閾値をいじることによりどこから黒でどこから白にするか、というのを決めることができます。これにより、閾値を下げていくと白に近い色から透過していき、逆もまた然りです。この閾値をトラックバーで自由に調整することができるようにしてあるので、透過色を指定することなく閣下画像を出力しながら自分で調整することで疑似的なクロマキー合成を可能とするのです。例えば人間を背景画像の前面に表示させたいならば人間の肌の色の補色である青色をバックに映像をとりこむといいでしょう。

4　今回作成したコード

作成したコードのメイン部を紹介します。環境は VS2013、言語は C++ です。

```
1  #define _USE_MATH_DEFINES
2  #include <iostream>
3  #include <cmath>
4  #include <opencv2/opencv.hpp>
5  #pragma comment(lib,"opencv_imgproc244d.lib")   // OpenCVライブラリ
6  #pragma comment(lib,"opencv_core244d.lib")      // OpenCVライブラリ
7  #pragma comment(lib,"opencv_highgui244d.lib")   // OpenCVライブラリ
8  #pragma comment(lib,"opencv_video244d.lib")     // OpenCVライブラリ
9  #pragma comment(lib,"opencv_objdetect244d.lib") // OpenCVライブラリ
10
11 using namespace std;
12 using namespace cv;
13
14 string win_src1 = "src1";
15 string win_src2 = "src2";
16 string win_dst = "dst";
17 string file_src2 = "bushitsu.png"; // 背景画像のファイル名
18
19 Mat img_src1 = imread(file_src2, 1); // 背景画像の読み込み
20 Mat img_src2;
21 Mat img_dst;
22 int value;
23
24 void onTrackbarChanged(int thres, void*) {
25     value = thres;
26 }
27
28 int main(int argc, char **argv) {
29     string file_src = "aru.jpeg"; // 入力画像
30     string file_dst = "dst.jpeg"; // 出力画像
31
32     VideoCapture capture(0);
33     if (!capture.isOpened()) {
34         cout << "error" << endl;
35         return -1;
36     }
37
```

```
38      // ウインドウ生成
39      namedWindow(win_src1, CV_WINDOW_AUTOSIZE);
40      namedWindow(win_src2, CV_WINDOW_AUTOSIZE);
41      namedWindow(win_dst, CV_WINDOW_AUTOSIZE);
42      createTrackbar("thresh", "dst", &value, 255, onTrackbarChanged);
43      setTrackbarPos("thresh", "dst", 150);
44
45      Mat  msk, msk_n, cut1, cut2;
46
47      while (1) {
48          capture >> img_src2;
49
50          cvtColor(img_src2, img_dst, CV_RGB2GRAY);
51          GaussianBlur(img_dst, img_dst, Size(11, 11), 3);
52          threshold(img_dst, img_dst, value, 255, THRESH_BINARY);
53
54          vector<Mat> channels;
55          channels.push_back(img_dst);
56          channels.push_back(img_dst);
57          channels.push_back(img_dst);
58          merge(channels, msk);
59
60          // 切り出し画像1
61          bitwise_and(img_src1, msk, cut1);
62          // マスク画像の反転
63          bitwise_not(msk, msk_n);
64          // 切り出し画像2
65          bitwise_and(img_src2, msk_n, cut2);
66          // 切り出し画像1と切り出し画像2の論理和
67          bitwise_or(cut1, cut2, img_dst);
68
69          imshow(win_src1, img_src1);    // 入力画像を表示
70          imshow(win_src2, img_src2);    // 背景画像を表示
71          imshow(win_dst, img_dst);      // 出力画像を表示
72          imwrite(file_dst, img_dst);    // 処理結果の保存
73
74          if (waitKey(1) == 'q') break; // q キーで終了
75      }
76      return 0;
77  }
```

5　おわりに

いかがだったでしょうか。これを機にできる限り多くの人に OpenCV に興味をもって貰いたいです。このプログラムは素人細工でまだまだ改善の余地が多いです。ぜひとも皆さんの手でよりよいものにしていってください。

Twitter: @Tachiazul

2016 年 07 月 20 日

マルコフ連鎖で二次元美少女の召喚を試みた話

h1dia

1　はじめに

　あなたは恋愛アドベンチャーゲームをプレーした事はありますか？そうです。パッケージに二次元美少女が 4 人くらい印刷された、四角い箱で 1 つ 7000 円くらいするゲーム群のことです。このようなゲームではキャラクターと甘いひとときを過ごす事ができますが、いつまでも会話することは出来ずどこかでエンディングを迎えてしまう、という問題が挙げられます。また、当たり前ですがシナリオは一定のため、何度プレーしても画面の中のキャラクターは同じ挙動を起こします。私はこのような恋愛アドベンチャーゲームをプレーするのですが、いつもエンディングを迎えると達成感とともに一種の喪失感を感じ、もっと二次元美少女とイチャイチャしたぁ〜ぃ！！と感情を爆発させたまま部屋の壁に頭をぶつけてしまいます。このままでは怪我をしますし、近所迷惑となってまうためこの問題はいち早く解決する必要があると考えました。そこで、今回はマルコフ連鎖を用いて文章生成を行うことで擬似的にキャラクターのセリフを生成し、そこに意思を持った二次元美少女がいるかのような錯覚を起こすプログラムを作成することで、いつまでも二次元美少女と甘い時間を過ごせるようにして私の部屋の壁の破壊を防ぐことを目標とします。

2　マルコフ連鎖とは

表題にもある「マルコフ連鎖」とは、何なのでしょうか？例えば、コミックマーケットの待機列で並ぶ人たちはこのような状態遷移のモデルで表すことが出来ます。

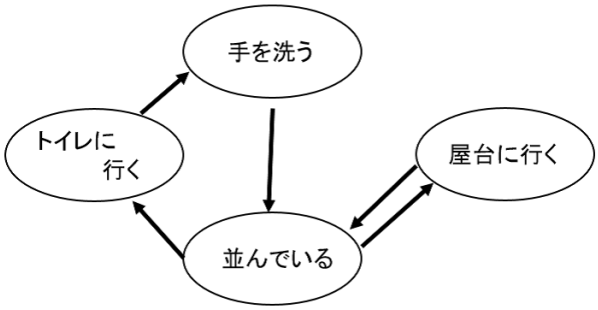


図１　コミックマーケット待機列に並ぶ人達のマルコフ連鎖

参加者は待機列で並び続け、トイレに行きたくなったトイレに行き、その後手を洗って戻ってきます。たまに手を洗わないこともあるでしょう (やめてね)。 ここでポイントになるのが、この人たちは「現在の状態のみを判断して次の行動を起こしている」という点です。未来の状態を予測したり、いままで辿った状態については勘案していません。

このような「現在の状態のみで未来の状態が決定する」ようなモデルをマルコフ過程といい、図のような飛び飛びの状態 (離散状態) のマルコフ過程をマルコフ連鎖といいます。

これを自然言語に当てはめて考えてみましょう。文章を考えます。

```
1  これはリンゴ
2  あれは何
```

この二つの文章を形態素にばらすと、

```
1  これ　は　リンゴ
2  あれ　は　何
```

となります。ここで、どちらの文章も「は」という副詞で単語が接続されています。この 2 つの文章から「は」の次には「リンゴ」「何」という 2 つの単語が現れることがわかりました。すなわち、「〜〜は○○」の○○に相当する部分には「リンゴ」「何」が入っても良いことになるため、これを先に挙げた状態遷移とみなすと

```
1 これ　は　何
2 あれ　は　リンゴ
```

というような、新たな文章をマルコフ連鎖によって生成することが出来ます。マルコフ連鎖はこのように文章を確率的なモデルで表すため、非常に簡易な方法で実装することが利点として挙げられます。今回はこのような方法を利用して文章を生成することを目標とします。

3 問題点

しかし、このようなマルコフ連鎖では表象文字についてのみ考えています。例えば、次の文章では「で」という文字については異なる用法で用いられていることが分かります。

ex) 「で」、どうする？ (接続詞的用法)
　　これ「で」、大丈夫です (副詞的用法)

先に挙げた「は」の用法はどちらも副詞的用法ですが、こちらの「で」の用法はそれぞれ異なります。実際に文章を組み替えてみると「で、大丈夫です」「これで、どうする？」と、2 文章の単純なモデルであるのにもかかわらず既に文章の意味が通っていません。これは文字単位に元の文章の品詞情報が欠落してしまっていることが原因であるため、このような文字の持つ意味を保存する必要があるといえます。

4 実装および結果

詳しい実装方法はかなり長くなるため、割愛します (下記のブログに詳細を記載していますのでご参照ください) まずは MeCab 等の形態素解析ツールによって文章を形態素ごとに分かち書きを行います。この際、文章とセットで形態素の品詞情報を取得します。この得た品詞情報と形態素を 1 対 1 対応で扱うことで解決します。具体

例として、「で (副詞)」「で (接続)」のような、得られた品詞情報をそのまま形態素に書き込んだものをマルコフ連鎖することで、表象文字の持つ品詞情報の欠落を防ぎました。

実際にゆずソフト「サノバウィッチ」のテキストから作中のキャラクター「因幡めぐる」ちゃんのセリフをうまく抽出し、上記の方法でセリフ文章に対してマルコフ連鎖を行った例を下記に示します。

```
1　っ！センパイわざとでしょ！）
2　細かいなぁ……
3　え？こんなところもありますよね？センパイ？！？こういうわけじゃないですけどぉ……長年の癖が
4　センパイ、いいじゃないですけど
5　い、ケーキ食べに行くとかよく貸し借りしたんだろ。こんないい人が、はい。センパイは雪でも……
6　うぅ……気になる方法なんて返せば！だ……なんて！トイレの個室狭いし、実際にですか？あ、こうやってるなら、あそこです。もう着替えはないですし……）
7　センパイって……あ、本当ですかと思いますよ……では、その……
8　ありがとうございます、みたいな人だったら、サイテーですよね……してあげるんです！そうですよ、センパイに何度も何度もメールしてもー！
9　それだけ頑張って下さい！
10　あー、はい！せっかく自分、寧々先輩とか……
11　あっ、さっき……はい……ど、シースーって、さっき一緒に！
12　は、んんぅ、あの……仮屋先輩と私、感じられるなんて、悪いわけで、うん……恋、今日の話にしないって
13　マグロとか…
14　は、はい！のどに溶け込めてるからですから
15　しょーが好きそうじゃない！その点で間に合ってたらお待たせて、頭からぱんつを……？
16　女子同士の悩みなんかある展開のヴァンパイアコス、恰好をしたことですよ！？自分はあのセンパイにおめでとう、センパイくすぐったいですよう……
17　あ、寧々せんぱぁっ……
```

ここに挙げた多くの文章は意味の通る文章とは言い難いです。それはマルコフ連鎖の性質によるもので、生成された文章に対して文法が正しいか、また単語同士の意味の係り受け等の評価を一切していないためです。しかし、文章全体ではなく形態素の前後だけを見ると 3 節で挙げたような、短文の状態で意味の通らない文章が見受けられません。したがって、品詞情報の欠落によってめちゃくちゃな文章が生成されなく

なったことがわかります。

別件ですが、この生成された文章からは因幡めぐるちゃんの口癖、喋り方が全体的に現れています。文法的な要素はともかく、なんとなく二次元美少女の意思の欠片を掴むことが出来たような気がします。

5 おわりに

自然言語処理による文章生成にはまだ課題が多く残っているといえます。人間が普段、意識せずに用いる言語というものを正確にアルゴリズム化することが出来ないと言うのはとても不思議な事です。同時に個人の人格を文章に反映させることはより難しく、生物がロボットに取って代わることはまず不可能なのではないか、と私は感じました。それでも僕は画面の中のキャラクターと会話することが将来あると信じ、情報科学がより発展することを願っています。部屋の壁は壊れました。

Twitter: @h1dia
Blog: 因幡のミックスジュース http://h1dia.hateblo.jp
2016 年 07 月 20 日

逆ポーランド記法による数式パーサ

このは

1　はじめに

数式パーサと聞くと大変難しいものを想像するだろう。しかしながら、単純な数式をパースするだけならとても簡単である。複雑な数式をパースする場合は、構文解析ライブラリと言われるような Boost.Spirit.Qi などを使う必要があるが、例にあげたライブラリは非常に難解であるが、単純なものなら自分自身でも作ることができる。今回は逆ポーランド記法を利用した数式処理と、中間記法から逆ポーランド記法への変換パーサを解説したいと思う。

数式パーサを利用することでできることは、ゲーム分野では多いと考えられる。数式をパースできるということは、数式を使う部分は全て外部ファイルに置くことができる。したがって、ゲーム分野ではウィンドウの配置や、敵の移動、キャラのアニメーション、攻撃時のダメージ算出など様々なところでの利用ができる。

変換を解説するにあたり、まず数式の記法について解説する必要があるため、次項で解説する。

2 数式の記法

2.1 中間記法

中間記法は、普段私達が使用している数式の書き方そのものである。名前の由来は、『5 + 4 / 2』など数字の中間に演算子を記述するからである。

中間記法は、人間が理解しやすい。しかしながら、コンピュータで扱おうと思うと中々に厄介である。これは、演算子の操作対象が前後にあるために、次の演算子まで読み込まないと操作対象が全て把握できないことに起因する。また、上の数式の例で言えば、『+』の後の演算子は『/』であるが、これは『+』よりも優先的に処理される。したがって、最初に『4 / 2』をするために次の演算子まで更に読み込まなければならない (この例では、次の演算子が存在しないから末尾まで読み込む)。

このように、中間記法はコンピュータで扱うのに適していない。ではどのように扱えばいいのかと問われれば、次項で解説する逆ポーランド記法を使用するのが好まれる。

逆ポーランド記法

逆ポーランド記法は、演算子の前方に操作する対象の数値を置いたものである。中間記法でも利用した『5 + 4 / 2』では、逆ポーランド記法で書くと『5 4 2 / +』となる。

逆ポーランド記法は、プログラム上では『スタック』を用いることで容易に処理することができる。数値は見つけたら全てスタックに積む。演算子が見つかった場合は、スタックから 2 つ数値を取り出し、計算した結果を再びスタックに積む。これを数式全ての文字が処理し終えるまで行うと、最終的にスタックに 1 つだけ値が残る。この値が、数式の答えとなる。

逆ポーランド記法は中間記法よりもプログラムでは非常に容易に書ける。しかし、外部ファイルに置いている数式は私達が理解しやすい中間記法で書く方がよい。したがって、中間記法から逆ポーランド記法への変換が必要となる。次項でこの変換についての解説をすることとする。

3　中間記法から逆ポーランド記法への変換

3.1　コードの解説

以下に載せるソースコードでは、RPN という class に SetExpression というメンバ関数がある。また、private なメンバ変数 element_があり、この型は`std::vector<std::string>`である。for 文で数式を回すのだが、『(』と『)』があった場合、この内部には数式が含まれている可能性があるために、再帰的に SetExpression を呼び出さなければならない。また、『(』と『)』の内部は 1 つの数式であるからこの中はこの段階の for 文では処理するべきではないために、indent という変数を用いて制御している。

RPN::GetCharValue(chare) は、文字が『+』と『-』であれば 1 を返し、『*』と『/』であれば 2 を返し、それ以外では-1 を返す関数である。つまり、演算子が入ってきた場合はその優先度を返し、数値であれば-1 を返す関数となる。

演算子は、見つかった場合に stack という`std::stack<std::pair<char,int>>`型の変数に push される。その際に、まず stack の一番上を参照し、入れようとしている演算子と一番上の演算子の優先度を比較し、入れるほうが優先度が高ければそのまま push する。しかし、一番上の方が優先度が高い場合は、入れる方の優先度が、一番上の優先度よりも高くなるまで、stack 内を一番上から element_に push する。そして、最後に stack に push する。こうすることで、演算子の順番が正しく変換できる。

数値は、演算子が見つかるまでは std::string 型の value 変数の末尾に push されていき、演算子が見つかった段階で element_の末尾に value 変数の値が push される。これで、数値はパースできる。

以上で、中間記法から逆ポーランド記法への変換はできたから、次項で逆ポーランド記法の処理を解説する。

3.2　ソースコード

```
1  RPN& RPN::SetExpression(const std::string & Exp) {
```

```
2     std::string value;
3     std::stack<std::pair<char, int>> stack;
4     int indent = 0;
5     for (auto&& str: Exp) {
6       if (str == '(') {
7         ++indent;
8       }
9       else if (str == ')') {
10        --indent;
11      }
12      if (indent == 0) {
13        int ch = RPN::GetCharValue(str);
14        if (ch != -1) {
15          if (value.front() != '('&&value.back() != ')') {
16            this->element_.push_back(value);
17          }
18          else {
19            this->SetExpression(value.substr(1, value.size() - 2));
20          }
21          value.clear();
22          if (stack.empty()) {
23            stack.push({ str,ch });
24          }
25          else {
26            if (stack.top().second < ch) {
27              stack.push({ str,ch });
28            }
29            else {
30              while (!stack.empty() && stack.top().second >= ch) {
31                this->element_.push_back({ stack.top().first });
32                stack.pop();
33              }
34              stack.push({ str,ch });
35            }
36          }
37        }
38        else {
39          value += str;
40        }
41      }
42      else {
43        value += str;
44      }
```

```
45    }
46    if (!value.empty()) {
47      this->element_.push_back(value);
48    }
49    while (!stack.empty()) {
50      this->element_.push_back({ stack.top().first });
51      stack.pop();
52    }
53    return *this;
54  }
```

4　逆ポーランド記法のプログラム

4.1　コードの解説

以下に載せるソースコードでは、上で解説した変換のコードと同じように RPN という class 内に、CalcExp() というメンバ関数を作っている。スタックから 2 つ値を取ってくるという処理が共通であるから、Lambda 式を利用して簡単に呼び出せるようにしてある。

やっていることは非常に単純で、element_を range-based-for で回し、演算子であればスタックから 2 つ値を取り出し計算し、スタックに積む。数値であれば、std::string 型から double 型に変換してからスタックに積む。そして全て回し終わったらスタックの一番上を返す。ただこれだけである。C++11 を使用しているから、見たことがないものもあるかもしれないが、やっていることはこれだけなので、解説はここまでとする。

4.2　ソースコード

```
1   double RPN::CalcExp() const {
2     if (this->element_.empty())return 0.0;
3     std::stack<double> value;
4     auto func = [&]() {
5       auto v1 = value.top(); value.pop();
6       auto v2 = value.top(); value.pop();
7       return std::make_pair(v1, v2);
```

```
 8    };
 9    for (auto&& el : this->element_) {
10      if (el == "+") {
11        auto pair = func();
12        value.push(pair.first + pair.second);
13      }
14      else if (el == "-") {
15        auto pair = func();
16        value.push(pair.second - pair.first);
17      }
18      else if (el == "/") {
19        auto pair = func();
20        value.push(pair.second / pair.first);
21      }
22      else if (el == "*") {
23        auto pair = func();
24        value.push(pair.first * pair.second);
25      }
26      else {
27        value.push(std::stoi(el));
28      }
29    }
30    return value.top();
31  }
```

5 まとめ

上で述べた数式パーサは、まだ変数や数学関数に対応していない未完成なものである。しかしながら、上のプログラムを改良すれば比較的容易に対応することができるであろう。たったのこれだけで数式がパースでき、数式の答えを得ることができる。これによって、外部ファイルに数式を書くことができ、実装が楽になる。

解説にもなっていないような拙い文章であったとは思うが、許してほしい。また、上のソースコードはリファクタリングする時間がなかったため、汚いものとなっている。しかしながら、要点だけ掴めれば自分でコードは書けると思う。ぜひ試して、色々なものをパースしてほしいと思う。

2016 年 7 月 17 日

ハフマン符号の複号

トド

1　はじめに

最近ハフマン符号を復号する機会があったので、その方法を記したいと思う

2　ハフマン符号とは

圧縮アルゴリズムの一種であり、主に同じようなデータ値を多く含むデータの圧縮に用いられる

3　ハフマン符号の仕組み

例えばデータ値が 0〜9 で構成された次のようなデータをハフマン符号で圧縮するとする

01 01 04 05 01 04　08 01 00 03　02　01 09 01 09　04　01　01　04

　すると

01 08 01 01 04 01 00 00 01 02 3D BE 22 A5 AD E7

　と圧縮できる

3.1 解説

まず、圧縮されたデータの最初の 10 バイトはデータ値の数を表しており
00 が 1 個、01 が 8 個... といった具合で 10 個のデータ数の個数が書かれている
まとめると下図のようになる (Large Small　ビット列 欄は後の図と合わせるため
空白にしている)

番号	数値	個数	Large	Small	ビット列
0	0	1			
1	1	8			
2	2	1			
3	3	1			
4	4	4			
5	5	1			
6	6	0			
7	7	0			
8	8	1			
9	9	2			

3.2 複号の準備

番号の低い方から 0 を除く個数の一番すくない数値を走査する、同じ個数の場合はスルー
そして一度すでに選んだ番号は無視し、もう一度同じ走査を行う
すると今回の場合前者が 0 と後者が 2 が当てはまる

新たに番号 10 の列を作り、個数の部分に先ほど選んだ二つの数値の個数の合計を個数の部分に
選んだ前者の番号を Small の欄に、後者を Large の欄に入れる
すると次のような表になる

番号	数値	個数	Large	Small	ビット列	選択済み
0	0	1			0b0	True
1	1	8				
2	2	1			0b1	True
3	3	1				
–	–	–	—	—	—	
9	9	2				
10		2	2	0		

二, 三回目も同じように行っていき、データ数以上の番号の新たに作った列も加えて行う

番号	数値	個数	Large	Small	ビット列	選択済み
0	0	1				
1	1	8				
2	2	1				
3	3	1				
–	–	–	—	—	—	
9	9	2				
10		2	2	0		
11		2	5	3		
12		3	9	8		
13		4	11	10		

そして同じように繰り返すと

番号	数値	個数	Large	Small	ビット列	選択済み
0	0	1			0b1000	True
1	1	8			0b0	True
2	2	1			0b1001	True
3	3	1			0b1010	True
4	4	4			0b111	True
5	5	1			0b1011	True
6	6	0				True
7	7	0				True
8	8	1			0b1100	True
9	9	2			0b1101	True
10		2	2	0		True
11		2	5	3		True
12		3	9	8		True
13		4	11	10		True
14		7	4	12		True
15		11	14	13		True
16		19	15	1		

という表が出来上がる

4 復号する

例に挙げている　01 08 01 01 04 01 00 00 01 02 3D BE 22 A5 AD E7 の後半部分 3D BE 22 A5 AD E7 を二進数表記にすると
00111101 10111110 00100010 10100101 10101101 11100111 となる
これを先頭 1bit 目から順に見ていくのだが
先ほどの表の最後の列から見ていく

bit が 1 の場合列の Large の番号に飛び、0 の場合 Small の番号に飛ぶ、そして飛んだ番号が数値の種類の数以下になった時の数値が圧縮される前のデータ値と決定

できる
まず最初は 0 なので 16 列目の Small の部分、番号 1 に飛び番号が数値の種類の数以下になったので数値は 1 と決定できる
二番目も同様に 1 と決まる
次に 1 となっているので 16 の Large の番号 15 に飛ぶ、次の bit も 1 なので 15 の Large の番号である 14 に飛ぶ
まだ数値の種類の数以下でないので次の bit も見ると 1 となっているので 14 の Large である番号 4 に飛ぶ
これは数値の種類の数以下なので数値は 4 と決まる
同じように繰り返していくと元のデータが複合できる。

5　最後に

今回データ値が 0 9 のみで構成されている場合を想定したが、データ値を 0xFF までもつ場合はデータの個数を表す部分を 256 個用意すればよい
例にとったデータは同じデータ値を多く持つのできちんと圧縮できたが、バラバラの値を持つ時、またはデータの部分が短い場合は圧縮効率が落ちる

マウスカーソルを変えよう

ねーつ

1　はじめに

最近マウスカーソルを変えて、おすすめしたいなと思ったのでそれに関して書きます。Windows 8.1 です。

2　カーソルって？

ああ！クリックする矢印みたいなやつです。この矢印は.cur や.ani ファイルとして PC に入っているものです。それを作るなり落とすなりして入れるとカーソルの画像を変えることが出来ます。

2.1　.cur,.ani とは

cur はいつもの矢印のような静止画です。他にホットスポット（クリックの座標）などの情報も入っています。ani は砂時計などの動くものです。cur を複数集めて作ったり出来ます。大きさは 32*32 です。

3　どうやって作るの？

これらは普通の画像ソフトでは開けません。なので専用のものが必要なのですが、フリーソフトがいくつかあります。合掌して落とします。許可についてよく知らないので、名前は伏せますが、RealW ● rld がいいなと思いました。

3.1　cur を作る

cur を作るのは簡単です。画像を 1 枚用意します。ここで透過されている png などだと楽です。横長や縦長だと見栄えが悪いかも？あと矢印のようなものを書かないとわかりにくい。これをソフトに読ませ、少し手を加えるなりして出力します。ソフトの中には、png 等を読み込み出来るものと出来ないものがあるみたいです。自分で描く場合ならばどちらでもいいですかね。

3.2　ani を作る

cur を複数読み込ませたり描いたりして作りました。絵が変わる時間等の編集することが出来ます。待ち時間のカーソルは動いていて欲しいですよね。しょうもないですが、例えば青い丸の回る速度を上げると、PC の処理が早くなったような気分になります。

4　どうやって変えるの？

いつものカーソル達は、C:￥Windows￥Cursors に入っていると思います。作ったものをそこに入れます。(C:￥cursor 等自分で作ってもいい。ただピクチャ等から参照すると再起動時にデフォルトになってしまうので、C 直下で)
参照するには、まずデスクトップの何もないところで右クリックすると出てくる個人設定に行きます。そこにマウスポインターの変更という項目があるのでそこからやります。変える対象がたくさんあります。

5 おわりに

読んで頂きありがとうございます。カーソルが華やかになると楽しいし見失いにくくなるのでいいと思います。ぜひおしゃれな物を作ってみて下さい。プログラミングで編集ソフト作成・・・とかは流石に無理です。どうやってつくるんでしょうね・・・

2016 年 7 月 20 日

自作パソコン（ゴミ箱）の製作記

いとしゅん（いっとあす、itus）

1　はじめに

こんにちは、もしくはこんばんは。いとしゅんと申します。入学してから 1 年が過ぎ、コンピ研の片隅におかせて頂いて 2 年目となりました。文化団体連合会の役員もやらせていただいてます。僕を一言で紹介させて頂くと嫌いなこと苦手なことプログラミングの趣味もパソコンがあまり入ってこないぐらいコンピ研の中で一番情弱です。正直他の皆さんの話についていけないです（なんで所属してるんだ...）
今回技術を何か語れとのですがホント語れることないです。（せいぜいガンプラのことぐらい←コンピューター関係ない）なので、春休みから 5 月の半ばくらいまでかけて自作パソコンを作ってみた時の話を少々してみたいと思います。
ほんとに話するだけなので読み飛ばし大歓迎です。（この枠潰しめ・・・）

2　友達に煽られて

なぜ作ろうとしたかと言うとまあ単純にデスクトップ欲しかったんですよね。
Minecraft の影 MOD くらいできればいいかなってレヴェルで、安いゲーミング PC でも買おうかと去年末あたりに考えてたんですけど（親との取り決めで長期休みしかバイトしちゃダメって言われてて）それを聞いた高校時代のデジタル大好き友人に「ん？お前コンピューター勉強してんだろ、自作パソコンくらい作らんのか？お？お？」（ウザさ 5 割増し）と煽られましてならば作るしかないじゃないかと。

そしたらその友人が彼の動画のネタにする条件付きで手伝ってくれると返事してくれ、気がついたら僕は春休み短期バイト社畜になってました。

3 実際に作った時のこと

前置きはこれぐらいにしておいて春休みせこせこ働いた分の資金を使った結果これらのパーツでミニタワー型を一つ作ることになりました。(パーツは友人が選んだので良い悪いとか全然わからんまま組み立ててました。)

○ CPU...Intel i5-6402P
○マザーボード...ASRock Z170M extreme4
○ CPU ファン...CoolerMaster HYPER 103
○ SSD...Sandisk SSD plus 120GB × 2
○メモリ...TED48GM2400C16DC0
○ケース...AEOS USB3.0
○キーボード...BUFFALO BSKBC02BKF
○電源... 玄人志向　KRPW-L5-500W/80+

その他ケースファン、配線や USB 有線マウス、win8、友人の食費（え？）、出張サービス代（ん？）などなんやかんやしめて 15 諭吉弱で制作しました。
Minecraft やりたいって言ってただけなのに予算内ギリギリまで（予算 12 万円くらい（笑)）かなり盛り込まれたことは素人の僕でもわかるものができました。結局僕が担当したのは物理的な組み立てと配線と、予算かせぎ、あと中身は、win10 アップグレードや初期設定と bios を言われるがままにいじっただけ？程度で OC のプラン提案やテストベンチ回しやソフトによる（なんかパソコンより高そうな）カスタムとかは友人が全部直接でもリモート（TeamViewer のちからってスゲー）でもやっていってしまいましたので、ここまで自作言っておいて中身のこと全然語れない他人作じゃねーか。はい、ページ無駄に使って申し訳ありません。(開き直るな)
でもマザーボードが初期不良起こして交換だとか OC の値間違えて暴走させたりこれ性能的は満足どころかヤバイよレベルなのに 5 分で落ちるカスタムとか実際にパーツを用意してみないと体験できない体験が得られたと思います。今までやってみたかったけど機会がなかったのでその点では友人に感謝してます。

学校でやってる授業よりこっち系のほうが好きだな。(ソースコード恐怖症ゆえ)

4　でもって使ってみた感じ

今までノートしかもってなかった僕の中では大げさかもしれませんがすごいスペック差を感じました。マイクラが快適だけじゃねえ、今までノートじゃ動かせなかったのが、こいつ動くぞ！って感じでできた瞬間は感動ものでした。コンピューター苦手マンでもずっといじってたいなあと思わずにはいられませんでした。現にこれも自作パソコンで作成してます。いやーデータ吹っ飛んでもやる気が萎えませんでした。5 分で落ちるカスタムのときは流石にテストベンチは超スペックな結果らしかったんですが（友人談）使うに堪えませんでした。(今は安定なとこ取れたのでバリバリ)ですが現実は非情である。学校の課題や試験が邪魔... ごほん。やりたいことばかりやってられるわけでもなく一日いじり倒せる時間がまだ取れてない状況です。(学生の本分おろそかにしてはいけない←お前成績下がってね？）夏休みはいじり倒したいです。設定使いやすくしたい。マイクラ建築したいね。

5　あとがき

自作パソコンのレポなのに肝心なところ知りたいところ載ってない、語ってない？ごめんなさい。そんなこと言われたらほんとに書くことないんです。他の皆さんや先輩のほうが絶対良い記事を書いてるに決まってます。ぜひそちらを読んで下さいおすすめです。もし僕のクソみたいな記事ここまで読んでいただいてる方がいらっしゃったらそれはそれでありがとうございます。すごい申し訳ない気持ちです。
さて、友人が挙げた動画なんですがそちらのほうで今回のパソコンについて詳しく解説しているのでちょっと紹介です。この動画は 5 分で落ちる頃の紹介ですが調整以外は完成したころの紹介になってます。ニコ動で sm28579500 で出てきたと思います。マイクラ MOD モリモリ激安ゲーミング PC をつくろうぜ!!という動画です。(動画の編集等は一切関わってないため紹介して良かったかは知りませんが一応)
最後になってしまいましたがこの冊子を手に取っていただきありがとうございました。
2016 年 7 月某日

main.cpp からソースファイルを分けるようになった男の話

まくら

1　はじめに

私は去年、ゲームプログラミングをファイルを分けずに main.cpp だけに書いて、ひどい目に合ったことを記事にしました。あれから 6 ヶ月、私はようやく失敗から学びソースファイルを分けるようになりました。オブジェクト指向プログラミングを意識したことにより、具体的にどう楽になったのかここに記述していこうと思います。

2　前回までのあらすじ

まずは前回、main.cpp だけにソースコードを書いたことにより、どういう問題が発生し、どう苦しんだのかまとめていきます。

- コードが延々と長くなっていくことによる混乱。
- 関数分けをしても今度はその関数の場所がわからない。
- どこからでもアクセスできるようにグローバル変数を使っているため、どこでどういった変化をしてるのか読みにくい。

この三点が主だった問題点でした。以下、この章はこれらがどのように解決していったかを説明していきます。

2.1　ショートショート

まず第一の問題点「コードが延々と長くなっていくことによる混乱」の解決について描いていきたいと思います。これは class を使うことによりすぐ解決しました。一つの要素、例えばボールであったら Ball という class を作りヘッダファイルとして設定することにより、それについての関数や変数が他のファイルにまとめられ、一つ一つのソースファイルが見やすくなっていきました。

2.2　押し合い圧し合いしない関数達

次に第二の問題点「関数分けをしても今度は関数の場所がわからない」は 2.1 で書いたように class 分けしていくことにより、method としてそれぞれの要素についての関数をまとめることによって解決されました。これによって関数が見やすくなり、どの要素に対する関数であるのかが格段に見やすくなりました。

2.3　小分けされていく変数

最後に第三の問題点である「どこからでもアクセスできるようにグローバル変数を使っているため、どこでどういった変化をしてるのか読みにくい。」というのは、やはり class を使うことでそもそもグローバル変数自体を使う必要がなくなり、さらに class 内で変数の動きがほぼ完結しているため頭が整理され、スムーズに開発できるようになりました。

3　さらなる長所

さらにオブジェクト指向で書いていくことにより以下のことが長所と実感として感じられました。それは、class を作成し、それを流用しつつ改良していくことで次回、次々回の開発が楽になっていくということでした。そのことによって、今後のゲーム開発のときさらなるクオリティアップが見込め、もっといい作品を作れるようになると感じました。

3.1 最後に

最後になりましたが「オブジェクト指向を学ぶことによりここまで楽しく楽にプログラミングできるのか。」という感想を最後に自分の記事を締めくくりたいと思いますここまでこんな記事を読んでくれた皆さま、ありがとうございました。次の機会があれば、もっとまともな記事を書きます。

2016 年 7 月 19 日

まくら

音声解析のすゝめ スペクトログラム編

フミノ

1　すべての始まり

「俺の声を内山昂輝[*1]さんみたいにしてくれ」
ある日、そうお願いされた。だからというわけではないが、その時期に信号処理に関して勉強していたので、これは知識と技術を活かすしか無いと思い[*2]音声解析について触れていった結果、見事にのめり込んでしまった。今回はその音声解析で用いられるスペクトログラムについて書いていきたいと思う。

2　短時間フーリエ変換

スペクトログラムについて説明する前に、短時間フーリエ変換 (以降, STFT) について触れなければならなりません。これはスペクトログラムを作成する方法の主な 1 つです。

[*1] 劇団ひまわり所属の男性声優 (2016 年 7 月 21 日現在). 出演作に『機動戦士ガンダム UC』 バナージ・リンクス役、『心が叫びたがってるんだ。』 坂上拓実 役などがある。
[*2] 本当は勉強しているうちに声優の声を解析して色々したいとも思った。

2.1 概要

STFT は離散フーリエ変換 (以降、DFT) を少ないサンプル数、短い時間毎に繰り返し行うスペクトル解析です。

2.2 フレーム長と分解能

DFT は解析する音のサンプリング周波数によって 1 サンプル当たりの時間が変わります。サンプリング周波数が 44.1kHz なら 1 サンプルは 1/44100=2*10^{-5} 秒です。何サンプル用いて解析するかによって周波数分解能は大きく変わります。このサンプル数をフレーム長と言います。サンプリング周波数が 44.1kHz なら、512 サンプルだと 86Hz で 1024 点だと 43Hz になる。FFT アルゴリズムを使うならこのサンプル数は 2 のべき乗にしなければならなりません。
フレーム長を長くすれば必ずしも良い解析になるとは限りません。DFT はあるサンプル数分を切り取って解析するため、そのフレーム内に複数の音が含まれているとそれが連続音なのか同時に鳴った音なのか判別がつかなくなります。連続音を一音ずつ解析するなら短いフレーム長で解析し、時間分解能をよくするしかないのです。短いフレーム長で時間分解能を良くし解析しようとすると周波数分解能が悪くなり、周波数分解能を良くしようとするとするとフレーム長が長く必要になり時間分解能が悪くなる。これをトレード・オフの関係と言います。

2.3 フレーム周期とパワースペクトル

DFT を繰り返し行うとき、フレームをある間隔で切り取り続けなければなりません。この間隔をフレーム周期と言います。では、この周期をどの程度にすればいいのか。単純にあるフレームの終わりからまたフレームを切り取れば良いのではっと考えますよね。しかし、DFT では切り取ったフレーム内の信号は周期的に繰り返してると考えて解析をし、一般に切り取ったフレームがちょうど信号の周期の始めと終わりに収まることはありません。そのため、周期的に繰り返しているとみなすた

めに切り取ったフレームに窓関数[*3]をかけてフレームの最初と最後を 0 に近づけ中央が最大になる信号にし、周期信号と見なして解析をします。
もし切り取るフレーム周期をフレーム長と同じ長さにした場合、窓掛けをしているのでフェードインとフェードアウトを繰り返している信号に変わってしまい元の信号とは大きく異なってしまいます。
ならばどの程度短くすればいいのか。窓掛けする窓関数にもよりますが、フレーム周期はフレーム長の 1/3 以下にするべきと考えられています。何故 1/3 以下なのか、フレーム周期がフレーム長の 1/2 の時と 1/3 の時のハニング窓を図 1, 図 2 に示します。

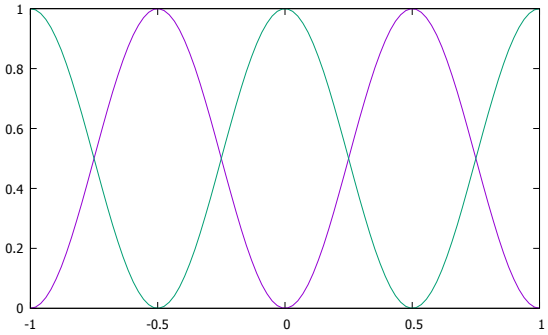


図 1　1/2 のフレーム周期でのハニング窓

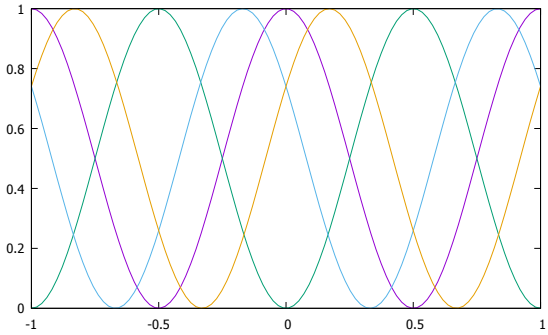


図 2　1/3 のフレーム周期でのハニング窓

これだけだと何処に問題があるのかわかりづらいと思います。では次にそれぞれ二乗してみた場合どうなるかを図 3, 図 4 に示します。

[*3] ハニング窓, ハミング窓, ブラックマン窓などがある

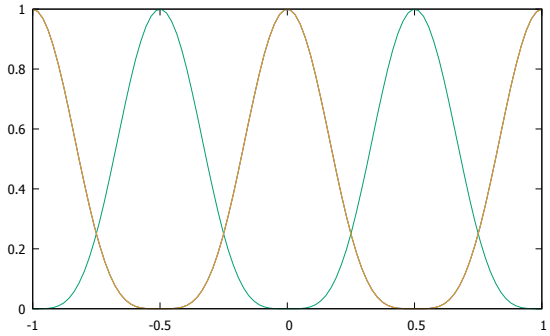


図 3　フレーム周期 1/2 のハニング窓の二乗

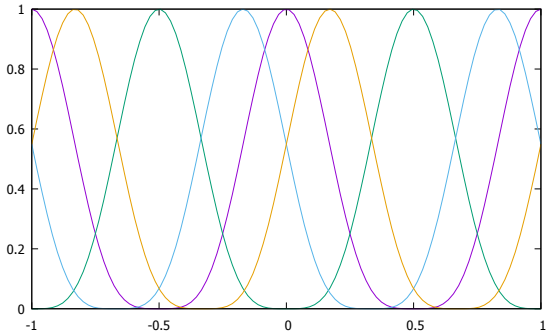


図 4　フレーム周期 1/3 のハニング窓の二乗

二乗の図ならわかりやすいと思います。1/2 のフレーム周期は交差してる点の値が小さいため、ここに掛かる信号の特徴が消えてしまいます。じゃあ二乗しなければいいのでは？っと思うかもしれませんが、そういうわけにはいきません。人は音を振幅ではなくエネルギー、パワーで聞いているため解析では各周波数の振幅では無くパワーで強さを考えなければいけません。振幅の二乗はパワーですので、二乗をする必要があるのです。

以上のことからフレーム周期はフレーム長の 1/3 以下が望ましい[*4]と考えられています。

人の聴覚が知覚できる範囲はとても広く最大と最小の比率は 100 万倍もあります。そのためパワースペクトルの値をそのまま使うと桁数が多くなるので、対数尺度をとります。この対数尺度をベル[*5][B] と言います。しかし、ベルはエネルギーが 10 倍で 1[B]、100 倍で 2[B] と小さい値なので使いにくいです。なのでベルの 10 分の 1 であるデシベル[*6][dB] を使います。ベル・デシベルは 2 つの値の比率なので基準となる値が必要です。この基準値は、使用用途で様々[*7]です。

[*4] 窓の形状に依存しますが、フレーム周期はフレーム長の 1/3、1/4、1/8 がよく使われます。

[*5] あるエネルギー E_a, E_b の対数尺度、ベルは $log_{10}\frac{E_a}{E_b}$

[*6] あるエネルギー E_a, E_b でのデシベルは $10log_{10}\frac{E_a}{E_b}$

[*7] 音圧を規格化したデシベル [dB SPL] や、電圧 1V を 0[dB] とした電圧の高さで表したデシベル [dBV] などがある。

2.4　高音強調

まずは図 5 を見てください。約 500Hz をピークにパワーは小さくなり、2500Hz から 6000Hz はバッサリと無くなっています。一般的に音声のスペクトルは全体的に傾斜で、高音の帯域はとてもパワーが小さくこのままだと解析できません。

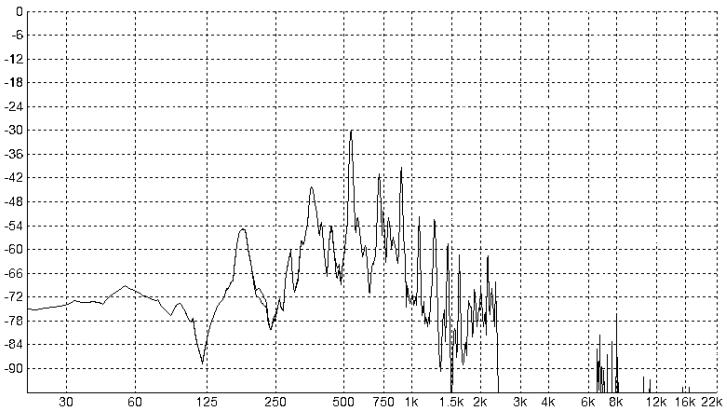


図 5　FFT で解析した結果

そのため高音部分を強めるために高域強調フィルタをかけます。今回は触れませんが、線形予測分析というものでも使います。
図 5 で用いた信号に高音強調フィルタをかけた場合の解析結果が図 6 です。図 5 では無くなっていた 2500Hz から 6000Hz、さらに高域の箇所にもスペクトルが出ました。
高域強調フィルタですが、これは差分フィルタでもあり離散信号だと

$$x_n = x_n - \alpha x_{n-1}$$

という式で実装できます。α の値はサンプリング周波数に依存しますが、0.9〜0.99 の値になります。

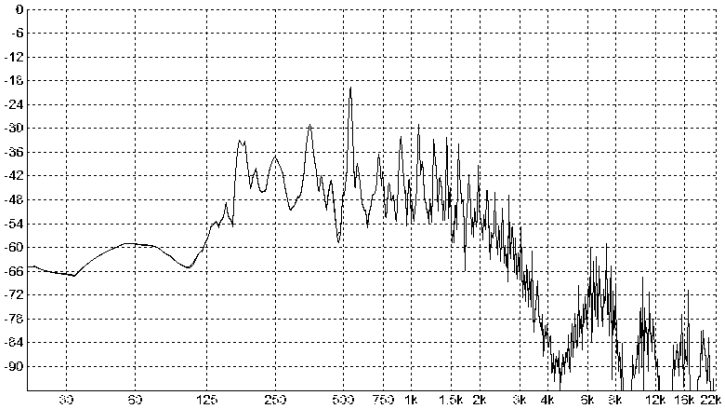


図 6　高域強調フィルタを掛けて FFT で解析した結果

3　スペクトログラム

本題のスペクトログラムです。とは言っても書くことは殆どありません。スペクトログラムとは、グラフの名前です。STFT での時間・周波数・パワーの三次元を使います。三次元のグラフですが、色の濃淡でパワーを表現することにして二次元上に落とし込み横軸を時間、周波数を縦軸にしたのがスペクトログラムです。
以下が STFT、スペクトログラムでのアルゴリズムです。

1．解析する音声から指定フレーム長で切り抜く
2．切り抜いた信号に窓掛け・高音強調を行う
3．FFT を行いパワースペクトルから各周波数のデシベルを求める
4．指定フレーム周期分ずらす
5．1〜5 を解析する音声の最後まで行う
6．得られた数値で色の濃淡を調整して描画

FFT のライブラリや描画のためのライブラリは色々あるので探してみてください。
今回自分は Siv3D を使って、スペクトログラムを作りました。

図 7 は夏川椎菜[*8]さんの声を解析した画面です。黒いほどその周波数が強く、時間での声の高さの変動がわかります。

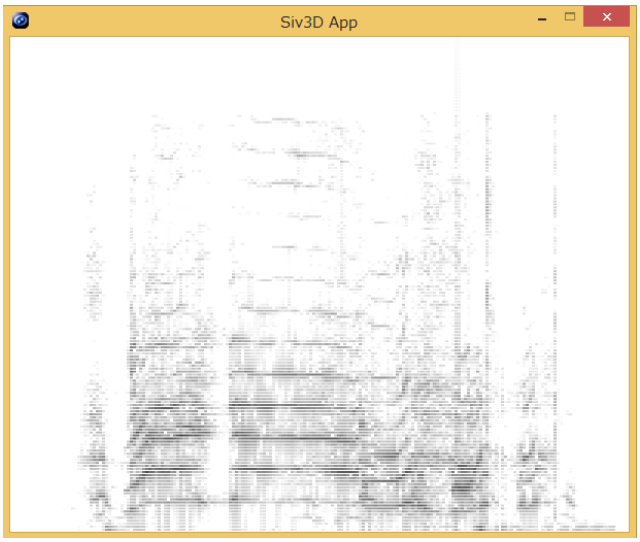


図 7　スペクトログラムでの解析表示

4　おわりに

音声解析の第一歩としてスペクトログラムを紹介しました。スペクトログラムはビジュアル的に一番わかりやすいのでオススメだと思い今回の原稿を書きました。声紋と呼ばれるものがこれなので、声紋解析もいつかしてみたいです。次は本文で言葉だけ出た線形予測分析について書ければなと思っています。
ライブに行きたい。

2016 年 7 月 21 日

[*8] ミュージックレイン所属の女性声優 (2016 年 7 月 21 日現在)。出演作に『ハイスクール・フリート』 岬明乃、『アイドルマスター ミリオンライブ!』 望月杏奈 役などがある。愛称はナンス (＊＞△＜) ＜ナーンナーンっっ

VisualStudio による C++ の co_yield,co_await の詳細

おいがみ

1　環境

- Visual Studio 2015 update 3
- プロジェクトの設定に/await が必要なので追加してください。
- co_await、co_yield はそれぞれ await、yield と書くこともできますが、ここでは co_await、co_yield と書きます。

2　とりあえず使ってみる

2.1　co_yield

```
1  #include <iostream>
2  #include <experimental/generator>
3
4  std::experimental::generator<char> f() {
5    for (auto& i : "abc") {
6      std::cout << '[';              // 1
7      co_yield i;                    // 2
8      std::cout << ']' << std::endl; // 4
9    }
```

```
}
int main() {
  auto obj = f();
  for (auto& i : obj) {
    std::cout << i;                  // 3
  }
  // for (auto& i : obj) { /* エラー */ }
  return 0;
}
/* output
[a]
[b]
[c]
[ ]                                  */
```

まぁ普通ですね。1,2 の処理の後に main 側の 3 に飛び、その後に 4 に戻っています。注意点としては obj を 2 回評価すると実行時エラーになります。
戻り値の std::experimental::generator<T> に関しては後で詳細を書きますが、auto を使うこともできます。co_yield があるので戻り値が std::experimental::generator<char>になります。ただし、VisualStudio の実装がバグっていて、i が const のときに std::experimental::generator<const char>になってしまうので注意が必要です。インテリセンスは正しい結果を表示しますがコンパイラがエラーになります。

```
#include <experimental/generator> // 戻り値が autoでも #includeは必要
auto f2() {
  for (auto& i : "abc") {
    co_yield i;
  }
}
```

2.2 co_await

work 関数は等差数列の和をループで計算しています。worker 関数は work 関数を std::future<T> にラップしています。co_await が使える条件に関しては後で書きますが、とりあえず std::future<T> を使っておけば大丈夫だと思います。co_await は戻り値に auto を使うことは現在できません。理由についても後述。

```
1  #include <iostream>
2  #include <future>
3
4  long long work(int a) {
5    long long ans = 0;
6    for (int i = 0; i < a; i++) {
7      ans += i;
8    }
9    return ans;
10 }
11
12 std::future<long long> worker(int a) {
13   return std::async(std::launch::deferred, [a]() {
14     return work(a); // 何らかの重い処理など
15   });
16 }
17
18 std::future<long long> run() {
19   std::cout << "begin" << std::endl;
20   long long ans = co_await worker(1000000000);
21   std::cout << "end" << std::endl;
22   return ans;
23 }
24
25 int main() {
26   using namespace std::chrono_literals;
27   std::future<long long> task = run();
28   for (;;) {
29     if (task.wait_for(500ms) != std::future_status::timeout) {
30       std::cout << task.get() << std::endl;
31       break;
32     }
33     std::cout << "wait" << std::endl;
```

```
34    }
35    return 0;
36  }
37  /*begin
38  wait
39  wait
40  wait
41  wait
42  wait
43  end
44  499999999500000000                  */
```

3 co_yield を使って filter, map, reduce を実装するサンプル

```
1   #include <experimental/generator>
2   #include <vector>
3   #include <functional>
4   #include <iostream>
5
6   auto create(std::vector<int> v) {
7     for (auto& i : v) {
8       co_yield i;
9     }
10  }
11  auto filter(std::experimental::generator<int> obj,
12                std::function<bool(int)> func) {
13    for (auto i : obj) {
14      if (func(i)) {
15        co_yield i;
16      }
17    }
18  }
19
20  auto map(std::experimental::generator<int> obj,
21             std::function<int(int)> func) {
22    for (auto i : obj) {
23      co_yield func(i);
24    }
25  }
```

```
26
27 auto reduce(std::experimental::generator<int> obj,
28               std::function<int(int, int)> func) {
29   int result = 0;
30   for (auto i : obj) {
31     result = func(result, i);
32   }
33   return result;
34 }
35
36 int main() {
37   std::vector<int> v(10);
38   for (int i = 0; i < v.size(); i++) {
39     v[i] = i;
40   }
41
42   std::cout << "filter" << std::endl;
43   auto is_even = [](int i) {
44     if (i % 2 == 0) return true;
45     else return false;
46   };
47
48   auto obj = filter(create(v), is_even);
49
50   for (auto& i : obj) {
51     std::cout << i << '␣';
52   }
53   std::cout << std::endl << std::endl;
54
55
56   std::cout << "map" << std::endl;
57   auto func = [](int i) { return i * 2; };
58
59   obj = map(create(v), func);
60
61   for (auto& i : obj) {
62     std::cout << i << '␣';
63   }
64   std::cout << std::endl << std::endl;
65
66
67   std::cout << "reduce" << std::endl;
68   int result = reduce(create(v), std::plus<int>());
```

```
69    std::cout << result << std::endl << std::endl;
70
71
72    std::cout << "filter␣&␣map␣&␣reduce" << std::endl;
73    result = reduce(map(filter(create(v), is_even), func),
74                        std::plus<int>());
75    std::cout << result << std::endl << std::endl;
76    return 0;
77  }
78  /*filter
79  0 2 4 6 8
80
81  map
82  0 2 4 6 8 10 12 14 16 18
83
84  reduce
85  45
86
87  filter & map & reduce
88  40                                  */
```

4 co_yield の詳細

戻り値として使える class の条件についてです。この例では型は int を返すものとします。

```
1   #include <experimental/coroutine>
2   struct my_generator {
3
4     struct promise_type {
5       int current_val;
6       my_generator get_return_object() { return my_generator(this); }
7
8       auto yield_value(int val) {
9         current_val = val;
10        return std::experimental::suspend_always{};
11      }
12      auto initial_suspend() {
13        return std::experimental::suspend_always{};
14      }
15      auto final_suspend() {
```

```
16        return std::experimental::suspend_always{};
17      }
18    };
19
20    using coro_handle
21            = std::experimental::coroutine_handle<promise_type>;
22
23    bool next() {
24      coro_.resume();
25      return !coro_.done();
26    }
27
28    int get() { return coro_.promise().current_val; }
29
30    my_generator(promise_type* prom)
31      :coro_(coro_handle::from_promise(*prom)) {}
32
33    ~my_generator() {
34      if (coro_) coro_.destroy();
35    }
36
37    my_generator() = default;
38
39    my_generator(my_generator const &) = delete;
40
41    my_generator &operator=(my_generator const &) = delete;
42
43    my_generator(my_generator &&right) : coro_(right.coro_) {
44      right.coro_ = nullptr;
45    }
46
47    my_generator &operator=(my_generator &&right) {
48      if (&right != this) {
49        coro_ = right.coro_;
50        right.coro_ = nullptr;
51      }
52      return *this;
53    }
54
55  private:
56
57    coro_handle coro_ = nullptr;
58  };
```

```
59
60 my_generator f() {
61   std::cout << "f" << std::endl;
62   co_yield 0;
63   co_yield 0;
64   co_yield 0;
65 }
66 int main() {
67   std::cout << "main" << std::endl;
68   auto a = f();
69   while (a.next()) {
70     std::cout << (a.get()) << std::endl;
71   }
72   return 0;
73 }
```

コルーチンはクラス内に promise_type クラスが必要です。更にその中に必要な関数が決まっています。これは範囲 for に begin() 、 end() が必要なのと同じです。

> Windows と C++ - Visual C++ 2015 におけるコルーチン (https://msdn.microsoft.com/ja-jp/magazine/mt573711.aspx#code-snippet-7)
> コンパイラは future をまったく認識しません。事実、future でなくてもかまいません。どういうしくみになっているのかというと、必要なバインディングをコンパイラを提供するアダプター関数を記述しないと機能しません。コンパイラは、範囲を基準にする for ステートメントの構造を把握する場合に適切な begin 関数と end 関数を探しますが、それに似ています。await 式の場合、コンパイラは begin と end ではなく、await_ready、await_suspend、await_resume のような関数を探します。begin と end と同様、これらの新しい関数は、メンバー関数でもメンバー以外の関数でもかまいません。メンバー以外の関数を記述できる機能は非常に便利です。

それでは一つずつ意味を紹介していきます。

4.1 my_generator promise_type::get_return_object() /* 必須 */

この関数は単純に promise_type の this を使って my_generator を生成して返してるだけです。一時停止もしくは関数を終了して呼び出し元に戻るタイミングで呼ばれます。

4.2 auto promise_type::yield_value(int val) /* 必須 */

co_yield i; とした時、 co_await yield_value(i); となります。戻り値に関しては initial_suspend() と同じです。
この関数でメンバ変数に保持することで、値を取得できるようになります。

4.3 auto promise_type::initial_suspend() /* 必須 */

co_await initial_suspend(); という処理が、関数の最初に入ります。
std::experimental::suspend_always は、 await_ready() が false を返すので一時停止されます。
std::experimental::suspend_never は、 await_ready() が true を返すので一時停止されません。
await_ready() の詳細に関しては 5.1 を読んでください。もちろん独自に定義することも可能です。これはサンプルを見たほうがわかりやすいと思います。

```
1  my_generator f() {
2    std::cout << "f" << std::endl;
3    co_yield 0;
4  }
5  int main() {
6    std::cout << "main" << std::endl;
7    f();
8    return 0;
9  }
10
11 // std::experimental::suspend_always の時
12 /* main */
13
```

```
14 // std::experimental::suspend_never の時
15 /*
16 main
17 f
18 */
```

4.4 auto promise_type::final_suspend() /* 必須 */

initial_suspend() と同じように、 suspend_always を返すと関数の終了時に一時停止処理が入ります。suspend_never を返すと、一時停止されずにコルーチンのメモリを解放します。

4.5 bool next()

coro_.resume() で一時停止を解除して次の co_yield もしくは関数終了に進みます。また、 final_suspend() で停止している場合に呼んだ時は、 std::terminate() が呼ばれます。
coro_.done() は final_suspend() で停止している場合に true を返します。なので final_suspend() が suspend_never を返してる場合は使えません。

4.6 int get()

単純に値を取得してるだけです。

4.7 my_generator(promise_type* prom)

コンストラクタですが、 coro_handle::from_promise(prom) で promise からハンドルを取得してるだけです。データのやり取り等はこのハンドルで行います。
get_return_object() 内で coro_handle::from_promise() を呼んでハンドルを渡す方法もありますが、どちらでも同じです。ただし参照渡しをする必要があります。コピーをすると正しく動かないので注意してください。

4.8 ~my_generator()

デストラクタではハンドルを使ってコルーチンの解放処理をしています。coro_.destroy() は一時停止しているものを解放しますが、すでに、解放されている場合に呼んだ時は未定義動作となります。なので、final_suspend() が suspend_never を返してる時に、関数の最後に到達してる場合はすでにメモリが開放されているので呼んではいけません。しかし、関数の最後に到達したか確認するすべもないので、必ず回数が決まっているものなどを除いて final_suspend() は suspend_always を返すことになると思います。

5 co_await の詳細

まず、初めに co_await を使う関数の戻り値に auto が使えなかった理由ですが、提案されている文書では、std::experimental::task<T> とされています。しかし Visual Studio には現在このクラスは実装されていません。なので使用することができません。Visual Studio では std::future<T> に対して、 bool await_ready(std::future<T>) 、void await_resume(std::future<T>)、 auto await_suspend(std::future<T>, std::experimental::coroutine_handle<> h) の 3 つが定義されているので代用ができるようになっています。**??**から**??**まではメンバ関数ではなく通常の関数として定義することも可能です。現に std::future<T>に対しては std 名前空間に定義されています。

```
1  #include <experimental/coroutine>
2  #include <future>
3  #include <chrono>
4  #include <iostream>
5  using namespace std::chrono_literals;
6
7  struct my_task {
8    struct promise_type {
9      std::experimental::suspend_never initial_suspend() { return{}; }
10     std::experimental::suspend_always final_suspend() { return{}; }
11     my_task get_return_object() { return my_task{ *this }; }
12   };
```

```
13    bool is_done() const { return coro_.done(); }
14
15    using coro_handle
16            = std::experimental::coroutine_handle<promise_type>;
17    my_task(promise_type& prom)
18      :coro_(coro_handle::from_promise(prom)) {}
19  private:
20    coro_handle coro_;
21  };
22  struct awaiter {
23    std::chrono::system_clock::duration duration;
24    awaiter(std::chrono::system_clock::duration d) : duration(d) {}
25    bool await_ready() const { return duration.count() <= 0; }
26    void await_resume() {}
27    auto await_suspend(std::experimental::coroutine_handle<> h) {
28      std::thread th([=]() {
29        std::this_thread::sleep_for(duration);
30        h();
31      });
32      th.detach();
33    }
34  };
35  template <class Rep, class Period>
36  auto operator co_await(std::chrono::duration<Rep, Period> d) {
37    return awaiter{ d };
38  }
39  awaiter my_sleep(std::chrono::seconds d) {
40    return awaiter{ d };
41  }
42
43  my_task f() {
44    co_await my_sleep(1s);
45    std::cout << "1秒待機" << std::endl;
46    co_await 1s;
47    std::cout << "1秒待機" << std::endl;
48  }
49
50  int main() {
51    auto task = f();
52    for (; !task.is_done();) {
53      std::this_thread::sleep_for(500ms);
54      std::cout << "wait" << std::endl;
55    }
```

```
56    return 0;
57  }
58  /*wait
59  1秒 待 機
60  wait
61  wait
62  1秒 待 機
63  wait                                                    */
```

co_yield とほとんど同じ my_task についての説明は省略して、 struct awaiter について説明します。

5.1 bool awaiter::await_ready() const /* 必須 */

true を返した時は、実行を中断しなかったものとし、 await_resume() を呼び実行を続けます。false を返した時は、 await_suspend() を呼びます。例では待機不要な 0 以下の時に true を返すようにしています。

5.2 void awaiter::await_resume() /* 必須 */

結果を取得する処理をします。今回は戻り値がないので何もしていません。

5.3 auto awaiter::await_suspend(std::experimental::coroutine_handle h) /* 必須 */

この関数が一番複雑です。 await_ready() が false を返した時に呼ばれます。戻り値を void にした時は必ず中断処理が挟まれます。また、戻り値を bool にした時は true の時に中断処理が挟まれます。中断時の再開処理は引数の h を使い h() を呼ぶことで、できます。例では void ですが、false を返すように変更すると、スリープされずに終了します。

5.4 auto operator co_await(std::chrono::duration<Rep, Period>d)

co_await 関数のオーバーロードみたいなものです。例では単純に awaiter にラップして返してるだけです。

6 co_await,co_yield を同じ関数で両方使う

標準では std::experimental::async_stream<T> が提案されていますが例によって VisualStudio にはまだ実装されていません。ですが、作ることはできると思うので時間があるときに書いてみようと思います。ちなみに戻り値を auto にして両方使おうとすると下記エラーが出ますが、ヘッダを含めても同じエラーになります。

```
1 error C3774: 'std::experimental::async_stream' が見つかりません。
2                <experimental/resumable> ヘッダーを含めてください
```

7 参考文献

- Windows と C++ - Visual C++ 2015 におけるコルーチン (https://msdn.microsoft.com/ja-jp/magazine/mt573711.aspx)
- Draft wording for Coroutines (Revision 2) [n4499] (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4499.pdf)
- Wording for Coroutines [p0057r0]] (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0057r0.pdf)
- Wording for Coroutines [p0057r4]] (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0057r4.pdf)

2016 年 7 月 23 日

MNVO 回線の速度比較

pandemo

1　目的

最近、MNVO 会社のスピードテストブーストなどが問題となっており、単にスピードテストを行うだけでは正確な通信速度を知ることができないことがわかった。なので実際にファイルをダウンロードするなどして、実測値を取ろうと思ったがすでにやっている人が多くいたため、あえて低速時の実測値を取ることにした。

2　原理

mineo, IIJmio の公式サイトで低速モードは下り 200kbps であると書いている。もしこのとおりであれば伝送率 100 %の時に 1.5MB のファイルを 1 分でダウンロードする事ができるはずである。

3　方法

Android 端末を PC に USB 接続してネットワーク接続して Google Drive から 1.5MB のファイルをダウンロードし、かかった時間を計測する。

4　使用したもの

mineoSIM MicroSIM
IIJmioSIM NanoSIM
SIM 変換アダプタ Mobile Labo
端末 ASUS ZenFone 2 Laser(ZE500KL)
PC LENOVO ThinkPad X250
使用サイト Google Drive
ダウンロードファイル 1 約 1.5MB(1575560B)
ダウンロードファイル 2 約 8.67MB(9093120B)
通信速度監視ソフト GlassWire

5　結果

5.1　表

表 1　1.5MB

回数	mineo	IIJmio
1	70s	55s
2	60s	60s
3	60s	55s
4	65s	65s
5	60s	60
平均	63s	59s
速度	23.80KByte/s	25.42KByte/s

どちらも思っていたよりも速度が出て差がわかりにくいので少し大きめファイルでも一度だけ試した

表 2　9MB

	mineo	IIJmio
	415s	340s
速度	23.80KByte/s	26.74KByte/s

5.2　考察

結果としては IIJmio のほうが mineo を僅かに上回る結果となった。　低速モード時の速度は 2 社とも 200Kbit/s = 25KByte/s で 1500KB ファイルのダウンロードには６０秒かかるので、実際には８０秒前後だろうと予想していたが意外にもほぼ理論値通りの結果が出て驚いた。というか IIJmio に関しては理論値を超えたスピードが出ているのでおそらく実際の値よりも少し遅めに公表しているのではないだろうか。

6　おわりに

IIJmio は計測中に 25KByte/s を超えることが何度かあり、多少の余裕を持たせることでユーザーにストレスを与えないようにするためだと考えられる。利用者が身近におらず、２社のみの比較となってしまったためもう少し多くの MVNO、特にユーザー視点の mineo, IIJmio とは対照的な MNVO である FREETEL, 楽天との比較を行ってみたい。

2016 年 07 月 13 日

GLSL をつかってみよう

yamacken

1　はじめに

OpenGL について勉強しようと「opengl 入門」だとか検索すると、まあ有名どころのサイトがたくさんみつかるわけですが。うんうん、こういう関数があるんだな！サンプルも読んだし、早速 OpenGL で開発だ！ なーんていざ使おうとしたときにあれ？僕の知ってる OpenGL と違う…といったことになるんですよね (実体験)

1.1　固定機能パイプライン

それというのも OpenGL 入門を謳った解説では大抵 OpenGL2.X に準拠した関数をメインにしていて GL2 ではポリゴンの表示とか光源処理も、画面に出力するまでの一連の処理を用意された固定機能を使うだけで勝手にやってくれていた（固定機能パイプラインと呼んだりする）のですが、バージョンの新しい OpenGL3.1 以降をはじめ、Android でも利用される組み込み機器向けの OpenGL ES2.0 や、three.js とかで流行ってる WebGL（OpenGLES2.0 がベース）のいずれでも大体の固定機能は取り除かれていて…
じゃあ固定機能を使って実装していた表現はどうすればできるんだ？という話になりますね。そこでででてくるのが GLSL なんですね

2 GLSL (OpenGL Shading Language)

いろいろ用語がありますが定義の説明なんか難しい上にどうでもいいので簡潔に言うと、GLSL という言語で描画に関する一連の処理の一部を自分でプログラムできるんだよという話です。(プログラマブルシェーダとか呼ばれてまる)
勝手に画像を転載するのはアレなので、ぜひ「opengl pipeline」と画像検索してもらいたいです。なにやら色分けされた四角形が矢印で繋がってる画像がたくさん出てきます。これらは OpenGL が画面に描画するまでの一連の処理をまとめた図で、渡されたデータに処理をして次の処理へ渡す動作が連なっているパイプライン処理をしているのがわかると思います。固定機能パイプラインのほうの図が混じっているので「vertex shader」という文字列のある図をさがしてそちらを見てください。気の利く図ならどこがプログラム可能か分かりやすいんですが、今回は VertexShader と FragmentShader というところの処理を書きます。

3 GLSL を使う

今回は開発環境を特別用意しないでも動かせる HTML5 Canvas+WebGL と GLSL ES とで使おうと思います。言語は JavaScript と html で書くことになります。ただ GLSL を使う流れは大体みんな同じなので他の言語での利用についても言及しようかなと思います

3.1 canvas をつくる

とりあえず html の body タグの中にそれっぽく記述していきます。

```
1 <canvas id="cv" width="500" height="500"></canvas>
```

JavaScript の関数で canvas を取得して、WebGL のコンテキストも取得します
あと body タグに onload とかつけて関数が実行されるようにしておいてください

```
1 <script>
2 function func(){
```

```
3         var cnvs = document.getElementById("cv");
4         var  gl  = cnvs.getContext("webgl");
5         ...
```

これで OpenGL のインスタンスを得たようなもので、JavaScript でない他の言語でも何かしら固有のやりかたがあると思います

3.2 GLSL のプログラムを読み込んで動くようにする

書いてもいないプログラムを読み込むなんてなんだか気持ち悪いですが、先に JavaScript にシェーダプログラムが利用可能になるところまで書いてしまいます

```
1 <script id="vs">バーテックスシェーダのプログラムをかくところ</script>
2 <script id="fs">フラグメントシェーダのプログラムをかくところ</script>
```

html ファイルの中に script タグで囲って GLSL のプログラムを書くことにします。id で javascript から取得できるようにしましょう

```
1   //実行される関数に追加
2     var vs_source = document.getElementById("vs").textContent;
3     var fs_source = document.getElementById("fs").textContent;
```

JavaScript だと変数の中身がわかりにくいですが、GLSL のプログラムの内容を文字列として取得させて持たせています (今だけソースと呼びます)。他の言語の型でいうと文字列を持たせられる変数 (配列) に入れます。とにかくソースをどうにか読み込めれば外部のファイルでもなんでもいいです。直接ガリガリ書いて代入しても動きます。

ここからはたった今取得したシェーダのソースと OpenGL(ES) の API を使って、いくつかの手順を経てプログラムが使えるようにしていきます。よってこのあたりはどの言語でも同じ GL の関数を同じ手順で使っていくことになります。

```
1   var vtShader = gl.createShader(gl.VERTEX_SHADER);
2   var frShader = gl.createShader(gl.FRAGMENT_SHADER);
```

まずはソースをコンパイルしたいんですが、そうするためのシェーダオブジェクトを作成します。引数はどのシェーダなのかを指定しています。返り値は整数でこれ自身を参照するためのＩＤとかハンドルだといえるものを返します。

```
1   gl.shaderSource(vtShader, vs_source);
2   gl.shaderSource(frShader, fs_source);
3
4   gl.compileShader(vtShader);
5   gl.compileShader(frShader);
```

上の関数でソースをシェーダオブジェクトに読み込ませ、下の関数でコンパイルさせています。さっきの返り値でシェーダオブジェクトを指定していますね。

```
1   var prog = gl.createProgram();
```

コンパイルしただけではまだ OpenGL の処理パイプラインには組み込まれていません。プログラムオブジェクトというものを作成してそれをやらせます。ここの返り値も id/ハンドルのようなものです

```
1   gl.attachShader(prog, vtShader);
2   gl.attachShader(prog, frShader);
3   gl.linkProgram(prog);
```

コンパイルしたあとのシェーダオブジェクトをアタッチしてリンクさせます。
何を言ってるのかよくわかりませんが、パイプライン処理でのある処理の入出力を別の処理とつなげることをしているらしいです。実際に中でどんなことをしているかは見えてきませんが、せっかくそこを気にしないで使えるようになっているので、こう書けばいいんでしょ程度で気にしないでいきましょう

```
1    gl.useProgram(prog);
```

この関数でプログラムオブジェクトがパイプラインに組み込まれました。これでやっと動くようになったんですね。
このプログラムオブジェクトの Id/ハンドルは後で使うので、3.2 節で書いてきた処理を関数にまとめたりしたときには無くしてしまわないようにしてください。
WebGL 特有の仕様かもしれませんが、gl のコンテキスト君に持っていてもらうこともできます

```
1   gl.program = prog;
```

3.3 シェーダのプログラムを書く

さっき飛ばした GLSL のプログラムを書いていきましょう。WebGL なので厳密には GLSL ES で、GLSL と若干違うところもあります。
書くまえに GLSL ES での修飾子を示しておきます

```
attribute 変数 : 頂点ごとに送られるデータ、頂点の座標、色, 法線など
                 vertex shader でしか使えない
uniform   変数 : すべての頂点に共通するデータ、変換行列など
varying    変数 : vertex shader からfragment shader への出力に使う
```

VertexShader のプログラムを書く

日本語で頂点シェーダです。html の script タグの中に書きます

```
1 <script id="vs">
2   attribute vec4 a_vertex;
3   varying vec4 v_color;
4   void main(){
5     gl_Position = a_vertex;
6     v_color = abs(a_vertex);
7   }
8 </script>
```

本来ならここで座標変換の処理をします (今回はやりませんが)。OpenGL を触った人ならなんとなく聞いたことがあると思いますが、なんたら座標系をうんたら座標系にする〜とかいうのです。そのために uniform 型の行列を変換行列として受け取って、それぞれの頂点を座標変換後の座標に変換していく処理をするのです。
5 行目に宣言してないのにいきなりでてきた変数がいますね。パイプライン処理なので、次のステージに処理したデータを渡さなくてはいけません。頂点シェーダで、その渡すデータにあたる出力は gl_position という組み込みの変数に入れるんです。入れるのは座標系変換した後の頂点です。
fragment shader で頂点ごとに違うデータを扱いたいときには、この頂点シェーダで varying を付けた変数に値を入れてあげます。

FragmentShader のプログラムを書く

日本語でフラグメントシェーダです。html の script タグの中に書きます

```
1 <script id="fs">
2   precision mediump float;
3   varying vec4 v_color;
4   void main()
5   {
6     gl_FragColor = v_color;
7   }
8 </script>
```

フラグメントシェーダは表示する図形の、画面上のピクセル一つ一つ（フラグメント）の色を決める処理をする。ピクセル一つ一つということなので、1 フレームごとにものすごい回数の処理がなされます。

2 行目に変なことが書いてありますが、ここで float の精度を中程度にしてね、と決めています。GLSL ES だけの仕様でこの記述がないとプログラムのリンクでエラーが起きるようになっているので、とりあえず書かなければいけません。組み込み機器向けの OpenGL ES では高精度の演算で演算効率が落ちるかもしれないことが理由らしいです。

3 行目には先ほど頂点シェーダで書いたものと全く同じ宣言がされています。頂点シェーダで入れた値がここで入力として読めるわけですね。

6 行目、ここでも組み込み変数がでてきています。今回はこれといった処理はしていませんが、gl_FragColor にはプログラムの処理で決定したフラグメントの色情報を入れます。

一番簡単な GLSL プログラムしか書いてませんがどうでしょうか。なんだか C 系の言語に書き方が似てますね (というか C 言語がベースです)。ここに頑張ってプログラムしていけばリアルな陰影や鏡面も表現できるんです。夢が膨らみますね。

3.4 表示させてみよう

やっと本命のここまでこれましたね。シェーダにデータをわたして簡単な図形を画面に表示させていきます。

ここからは JavaScript の関数に戻ってシェーダに頂点データを渡す処理を書いていきます。シェーダに入力するためのなんらかの手段が必要ですね。

```
1     var vert = [-1.0, 0.0,   0.0,1.0,  0.0, 0.0,];
```

その前に、当然ですが渡すデータは用意しなくてはいけません。

```
1   var vtBuffer = gl.createBuffer();
2   gl.bindBuffer(gl.ARRAY_BUFFER, vtBuffer);
```

シェーダにデータを渡すためにはバッファオブジェクトというものを使います。createBuffer でバッファオブジェクトを生成して、bindBuffer で ARRAY_BUFFER ターゲットとバインドしています。文字にしてもよく分かりませんが、なにをしたかというとバッファオブジェクトに頂点データを入れたい、と指定しただけです。

```
1   var vertices = new Float32Array(vert);
2   gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

先ほどの用意したデータを Float32Array という JavaScript の型付き配列に変えています。WebGL で JavaScript からデータを渡すときにはこの型を利用します。bufferData でさきほどバインドしたバッファにデータを書き込みます。3 つ目の引数は、OpenGL が内部でこのバッファを適切に管理できるように知らせてあげるヒントのようなものらしいです。

```
1   var vtloc = gl.getAttribLocation(prog, 'a_vertex');
```

使うといっていたプログラムオブジェクトがここで出てきました。頂点シェーダの attribute 変数の格納場所を取得しています。

```
1   gl.vertexAttribPointer(vtloc, 2, gl.FLOAT, false, 0, 0);
```

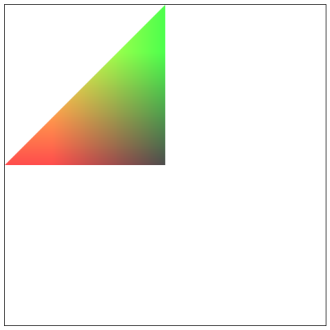
取得した格納場所に gl.ARRAY_BUFFER とバインドされているバッファオブジェクトを割り当てています。

```
1   gl.enableVertexAttribArray(vtloc);
```

いま割り当てたバッファオブジェクトの、割り当てを有効にします。これでバッファオブジェクトに書き込まれた頂点データが使えるようになりました。

```
1   gl.drawArrays(gl.TRIANGLES, 0, vertices.length/2);
```

これでやっっっと図形が表示されます。いやー長かったですね。
まったく同じプログラムを書いたなら、canvas の中心と、左辺の中点、上辺の中点を結んだ、頂点の色がそれぞれ違う三角形が描かれているはずです。



4 おわりに

GLSL をつかってみよう、とは言ったものの実際にこれだけページを使ってできたのほんの触りだけの Hello World 程度のものでしたね。
しかし、ここまで読んでくれた人ならわかってくれるでしょう。GLSL は最初の一歩を踏み出すだけでも、やらなければならないことが山のようにあるんです。勉強しよう、と思っても尻込みしてしまいますよね。
少しでもわたしの文章が役に立ったり、この先をもっと知りたい！と興味を持ってくれたらうれしいです。

2016 年 07 月 22 日

CADLUS X を使って基板を設計する

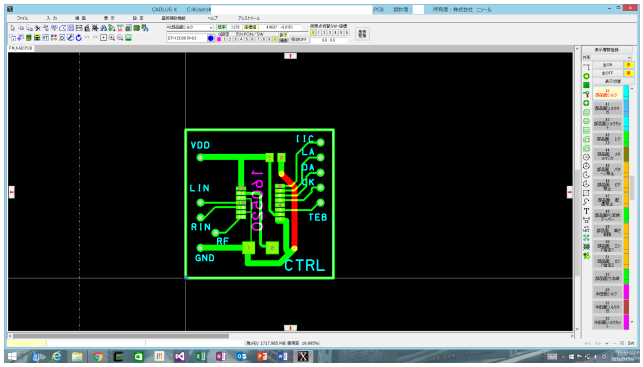
ミンクス

1　はじめに

これは、私がサークル活動において基板の作成にて行った手順についてまとめようと思い書きました。かなりの初心者なので至らないところもありますが、よろしくお願いします。
製作するときに参考にしていた書籍はオーム社の横田一弘 著「CADLUS+Arduino 電子工作ガイド」です。また、P 板.com のサイトの基板製作工程を紹介するページも参考にしました。

2　CADLUS X について

CADLUS X とは P 板.com の推奨してる CAD ソフトの一つで、P 板.com でアカウント登録することで無料で使うことができます。PDF で P 板.com の説明書が付いているが、自分にはかなり難しく感じました。



3　グリッド

CADLUS の機能で、一定間隔でクリックする場所を指定する機能があります。これは本に書いてなかったため気付くのに時間が掛かり、なかなか思うように設計できませんでした。上の数字でそれぞれのピッチがあらかじめ用意されています。「0」

でマウスカーソルの位置に好きに場所が指定できるようになります。

4　ランドの口径

差し込むピンの大きさちょうどで設計すると、ピンを差し込むときに苦労するので余裕を持たせる必要があるようです。

5　まとめ

全く知識のない状態からの基板の設計はかなり時間が掛かるものでした。今回わかった事は、ネットだけだとどうにかならないこともあるから自分に見合った本を見つけるということが分かりました。

ミンクス

2016 年 7 月 19 日

Ruby と Twitter の連携

トロ

1　あいさつ

こんにちは。去年の世田谷祭で bot 作ったのでそこらへんで使った知識で Ruby と Twitter を連携させる導入部分の話をします。

2　最初にすること

まずは`$ gem install twitter` で twitter ruby gem をインストール（？）します。次に使うアカウントの情報を入力します

2.1　初期設定

```
1     require "twitter"
2     require 'tweetstream'
3
4     client = Twitter::REST::Client.new do |config|
5       config.consumer_key        = "YOUR_CONSUMER_KEY"
6       config.consumer_secret     = "YOUR_CONSUMER_SECRET"
7       config.access_token        = "YOUR_ACCESS_TOKEN"
8       config.access_token_secret = "YOUR_ACCESS_SECRET"
9     end
```

Twitter と連携するのには Consumerkey, consumersecret, accesstoken,

accesstoken が必要です。これらはブラウザ版 twitter の設定から取得できます。 電話番号が登録されてないと取得できないのでそこは注意。
これで Twitter の API が使えます。だいたいなんでもできます。自分はメモツイートにふぁぼしてあとで呼ブ bot とか作りました。一時期お○がおチャレンジで話題になってた大量ブロックも多分これでやれます (真似してはいけない)。あとは診断メーカーっぽいこともできます。個人的にアンケート機能関連の API がないのではやく実装してほしいですね（もう実装されてるのかな）いろいろ面白いことができそう。

3　おわりに

Ruby 書きやすいので「俺は Twitter なんか興味ねえんだよ、二度とくるんじゃねえーよ！」みたいな人でも Ruby でなんかかいてもらいたいです。次の進捗は文章解析とかしてみたいけどなかなかいいアイデアが浮かばない、、、

Twitter: @torotoroleen
2016 年 7 月 22 日

あとがき

東京都市大学コンピュータ技術研究会です。1 年生が新たに入部し、3 年生は引退のため部誌を書くのも最後になってしまいました。いかがでしたでしょうか？そんなコンピュータ技術研究会の次回の学園祭、世田谷祭は 10/29, 30 に行われます。ぜひお越しください！

発行者：東京都市大学コンピュータ技術研究会
表紙：うろん
発行：2016 年 7 月 28 日
印刷：株式会社 栄光
Web： http://www.tcu-ctrl.jp/