TSG

Theoretical Science Group

理論科学グループ

※本作品はフィクションであり、人物名・団体名などは架空のものです。

部報 236 号
— 新入生自己紹介号 —

目　次

特集

新入生自己紹介

新入生の皆さん

毎年恒例の新入生自己紹介コーナーです。もうすぐ夏学期も終わろうとしているのに、「新入生」というのもアレな気がしますが、そこらへんは平に御容赦を。では自己紹介のはじまりまじまり～

50 音順になっているはずです

氏名	大野　寛子
科類?	早稲田大学第一文学部
出身校	千葉県立木更津高校
プログラミング経験	中 3 の時 BASIC をかじりそのまま忘れてしまいました。
得意技	下地処理
必殺技	塗装
抱負	栄養分が豊富な伊達巻になります。

氏名	岡野　諭
科類	理科 I 類
出身校	駒場東邦
プログラミング経験	VB が 6 年ぐらい。Java とか C/C++とかはほんのちょびっと。Perl は掲示板をもらってきて自分で設置できる程度。Ruby は興味あるけど経験はなしです。
得意技	適当な設計とパッチワークです。
抱負	これはやはりしっかりとした設計ができるようになることと Java をマスターすることでしょう。

新入生自己紹介

名前	菅原　悠 (スガワラ　ユウ)
所属	理科 I 類
高校	筑駒
プログラミング経験	中一より 7 年目。F-BASIC より始めて C,Java は一応使えます。
特異技	広く浅い知識をギリギリに使いこなす、誤変換および聞き違い。
必殺技	「どうしてくれよう!」など
抱負	ずっと dos や towns だったので、よその世界に羽ばたくことが目標です!

氏名	中嶋　海介
科類	理科 I 類
出身校	浅野
プログラミング経験	小 6 から HyperTalk (Mac) 中 1 から C/C++ (DOS) 高 2 から C/C++ (Linux,FreeBSD) で、ゲーム等をテキトーに作っておりました。
得意技	ピアノ/ギター (不得意ですけど (ぉ))
必殺技	マインスイーパ (上級 109 秒)
抱負	分科会に全然出てませんが (ぉ) 駒場祭には何か出します。

氏名	野村　芳明（HN:まんさく）
科類	理科 I 類
出身校	麻布
プログラミング経験	麻布パソコン同好会で中一〜中二に BASIC、中三〜高二に C をやってました。自作ゲーム多数（マリオ、ドラクエ、文化祭展示用など)。 SuperCon‘99 で準優勝です。
得意技	ゲーム（ゲー研にも入ってます)、パズル
必殺技	「ず」攻め（「ドラクエしりとり」にて）
抱負	昔ゲーム（遊ぶ方）目当てでパソコン部に入ってプログラミングだけ習ったためパソコン自体の知識が全然ないのでそっちの方をもうちょっと勉強した方がいいかなあと

氏名	塚本　多矩（「たく」と読みます）HN は”TAKU”又は”たく”です。
科類	理科 I 類
出身校	私立城北高校
プログラミング経験	皆無
得意技	善人のふりをすること（心はいつも狼です。）
必殺技	市中引き回しの上、打ち首獄門
抱負	今年の一年のなかで一番戦力になりそうにない男ですが、頑張ります。

氏名	牟田　秀俊
科類	理科 I 類
出身校	久留米大付設高校
プログラミング経験	C を 1.5 年
得意技	C マガ電脳クラブ
必殺技	多段ネスト
抱負	Win プログラムができるようになりたい。

氏名	本坊紘子
科類	文科 II 類
出身校	志学館高等部（鹿児島県）
プログラミング経験	なし
得意技	なし。
必殺技	なし。
抱負	得意技と必殺技の修得

氏名	師　芳卓（もろ　よしたか）HN:ROPPY,MUO
科類	理科 I 類
出身校	筑駒
プログラミング経験	中学のときは BASIC、高校のときは C 言語をやっていた。
得意技	ゲーム（主に RPG）
必殺技	円周率 100 桁を一瞬で言う
抱負	CGI や JAVA を学んで自分の HP をもっと良くし、ゲームもいろいろつくるつもり。

一般記事

ASP 2

Kit

Welcome to this crazy time

このイカレた時代へようこそ。

はじめに（遅

君は Tough Boy · · ·

というわけで 2 回目です。
前回何やるって言ったっけ…（ぉ

そうだ、「フォームデータを受け取る」をやるとか書いてました。これだけで 1 回分持たせるには量が少なすぎるので適当に付け足しておきました。

それでは

本文へどうぞ。

フォームデータを受け取ろう
～これで掲示板が作れる～

結論

`Request.Form(`*fieldname*`)` で取れます。

詳細

フォームのデータは ”フィールド名=値” の形で、フィールド名 (HTML で name=”hoge” とかやって指定した名前のこと) とその値が組になって送られてきます。複数組ある場合は ”&” を区切り文字として送られます。また、” ” (半角スペース) は ”+” に、半角英数[1]以外は ”%” に続く 2 桁の 16 進数に 1 バイトずつ変換されます。Perl の場合、送られてきたデータをフィールドごとのデータにするには

1. ”&” をデリミタに分割してフィールド名と値の組に分ける
2. ”=” をデリミタにしてフィールド名と値を分ける
3. `$value =~ tr/+/ /;`
   `$value =~ s/%([0-9a-fA-F][0-9a-fA-F])/pack("C",hex($1))/eg;`
   などとして元のデータに戻す

といった行程が必要になりますが、ASP(VBScript) においてはそんな操作は一切必要ありません (なら上に書いたことは何なんだ[2]) すべて `value = Request.Form("フィールド名")` でできてしまいます。サーバー様が上の処理を全部やってくださるのです[3]。

ちなみに、`Request.Form` が使えるのはリクエストが POST で送られてきた時だけです。つまり、元の HTML で `<form method="POST" action="〜">` となっている場合だけです。`method="GET"` となっている時 (検索サイトとかはこれですね) は URL の後に ”?” が続き、その後に ”フィールド名=値” の組が ”&” で区切られて出てきます。この場合にデータを受け取りたい時は、`Request.Form` でなく `Request.QueryString` を使います。使い方は `Request.Form` と同じです。

「いちいち使い分けるのめんどくさい」という私のような人にも ASP with VBScript はやさしいのです。何と、`Request(`*fieldname*`)` だけでデータが取れてしまいます。が、ちょっと注意点があります。Microsoft の IIS Help[4] から引用しますと、

> コレクション名を指定せずに `Request(`*variable*`)` という呼び出しを使用して変数に直接アクセスすることもできます。この場合、Web サーバーは次の順序でコレクションを探します。
>
> 1. QueryString
> 2. Form
> 3. Cookies
> 4. ClientCertificate
> 5. ServerVariables

[1] ”-” とかも含まれるかもしれません
[2] そんなこと言う人嫌いです。
[3] ビバ、コンピュータ様。いや、あの世界はほとんど知りませんが。
[4] http://www.microsoft.com/japan/developer/library/jpiis/iishelp/iis/asp/vbob5ulw.htm

> 同じ名前を持つ変数が複数のコレクションにある場合、Request オブジェクトは、最初に見つけたインスタンスを返します。

とのことです。要は QueryString と Form に同じ名前のフィールドがあった場合には QueryString の方が返る、ということです。

せっかくだから[5] QueryString, Form 以外の 3 つを説明しようと思ったのですが、Cookies は使ったことないし、ClientCertificate なんて初めて聞きました（ぉ ServerVariables だけ説明しますが、これは環境変数の値を取得するためにあります。`Request.ServerVariables("REMOTE_HOST")` とすればホスト名が、`Request.ServerVariables("HTTP_REFERER")` とすればリファラーが得られます。

話を戻して

フォームデータの取り扱いです。`<input type="checkbox">` や `<select multiple>` 等、複数選択を有効にした場合には、`Request.Form(`*fieldname*`)` で得られるデータは ”, ” (カンマと半角スペース) で区切られています（複数選択時）これを一つ一つのデータに分けるためには

```
var1 = Request.Form("fieldname")
var2 = Split(var1, ", ")
```

などとしてやればよいのですが、他の方法もあります。それを以下に記します。

```
For i = 1 To Request.Form("fieldname").Count
    var(i) = Request.Form("fieldname")(i)
Next
```

`var` は配列です。…配列って前回やったっけ？ ……やってないようです。

というわけで配列

配列を使う場合には宣言が必要です。宣言は以下のようにします。

```
Dim var(10)
```

`var` が配列名、10 が配列のサイズです。`var(4)` などとすれば配列内の要素にアクセスできます。インデックス (`var(4)` の時の 4) は 0 から始まります。C 言語などと違うのは、インデックスは 9 までではなく 10 まであるということです。そうです、`Dim var(10)` とした時には配列のサイズは実は 10 ではなくて 11 だったのです[6]。

[5] 俺はこの赤の扉を選ぶぜ

[6] 何でこういう仕様なんでしょうね。天下の Micro$oft 様のお考えになることは私のような下賤の者にはまったくわかりません。

では戻って

もう一度さっきのコードを載せます。

```
For i = 1 To Request.Form("fieldname").Count
    var(i) = Request.Form("fieldname")(i)
Next
```

`Request.Form("fieldname")` が配列みたいなものだということがわかりますね。`Request.Form("fieldname").Count` は `Request.Form("fieldname")` の要素数を返します。何をしてるかわかりますね。要素数分ループを繰り返してその都度変数に代入してるということは、配列に要素全部を入れた、ということです。しかし、ここで注意しなくてはならないことがあります。`For i = 1` という部分がありますね。そしてループ内で `Request.Form("fieldname")(i)` を参照しています。ということは、インデックス番号が 1 のデータから見ていることになります。なぜ 0 から見ないのでしょう。実は、さっき配列のところで「インデックスは 0 から始まる」と書きましたが、この場合はその限りではないのです (だから配列「みたいなもの」と言ったのですが) `Request.Form("fieldname")(0)` を参照しようとするとエラーになります[7]。注意してください。

これにて

フォームデータの受け取り方の解説を終えたいと思います。何か長々と書いた気がしても、書いてたことをまとめると「`Request.Form("field")` でデータが得られるのさっ」で終わってしまい愕然としている次第であります[8]。

というわけでちょっと延長して VBScript のデータ型についての話とかしたいと思います。

データ型

前回に「型」なんてない、と書きましたが、ちょっと違います。表面に出てくる型はありませんが内部処理形式に型はあります。それは、

> ブール型 (Boolean), バイト型 (Byte), 整数型 (Integer), 通貨型 (Currency), 長整数型 (Long), 単精度浮動小数点数型 (Single), 倍精度浮動小数点数型 (Double), 日付 (時間) 型 (Date), 文字列型 (String)

[7] 何でこういう仕様なんでしょうね。天下の Micro$oft 様のお考えになることは私のような下賤の者にはまったくわかりません。

[8] いやほんとに。

です。そしてたまには「文字列形式で (ダブルクォーテーションで囲んで) 数字入れたけど、この数にこっちの数を足してまた文字列型に戻したい」とか思う時があるわけで、そういうときには型変換を行います。型変換にはそれ専用の関数があって、上にあげた型に対してそれぞれ、

`CBool`(*variable*), `CByte`(*variable*), `CInt`(*variable*), `CCur`(*variable*), `CLng`(*variable*), `CSng`(*variable*), `CDbl`(*variable*), `CDate`(*variable*), `CStr`(*variable*)

です。

便利な関数

私のよく使う[9]関数を紹介します。

Server.HTMLEncode(*variable*)

この関数は *variable* を HTML ソースとして使えるように変換します。具体的には ”&” を ”&” に、”<” を ”<” に、”>” を ”>” に、ダブルクォーテーションを ”"” に変換します。

Response.Redirect *url*

url にリダイレクトします。

Len(*string*)

string の文字数を返します。

Left(*string*, *length*), Right(*string*, *length*), Mid(*string*, *start*[, *length*])

Left は文字列 *string* の左端から *length* 文字分の文字列を返します。Right は Left の右側バージョンで、文字列 *string* の右端から *length* 文字分の文字列を返します。Mid は *string* の *start* 文字目から *length* 文字分の文字列 (*length* が指定されていない時は *start* から後の全ての文字) を返します。

[9] というか、以前書いたソースを読んでみたらよく使っていた

String(*number*, *character*)

character に指定した文字 (または文字列の先頭文字) を *number* 文字分並べた文字列を返します。「数字の右側に 0 をつけて 10 桁にする」とかいったことをするときに役立ちます。

終わりに

We are living, living in the eighties···

いや、生まれたのが 1980 年だから間違っちゃいませんが。

さて、おしまいです。今回は量がかなり少ないですね[10]。やはり 1 日で書き上げるのは無理があったというものです[11]。次回こそは。

次回のネタですが、データベースアクセスとか書くかも。また少ない気がする…

しかし、この文章、需要あるんだろうか…？

We still fight, fighting in the eighties···

Linux カーネルを読もう

第一回: スケジューラ

菊

というわけで今回はスケジューラの話です。

Linux はマルチタスクをサポートした OS ですから，複数のプログラムが同時に走っています。この走っているプログラムのことをタスクとかプロセスと呼びます。カーネルはタスクのリストを持っていて，それを交互に実行することにより複数のタスクが同時に動いているように見せているわけです。

ひとつの CPU で実行できるタスクはひとつですから，カーネルはタスクのリストから動かすべきタスクをひとつ選んで，それを実行することになります。このタスクを選択するルーチンがスケジューラです。スケジュールを行う関数は `schedule()` という名前です (そのままですね)。

[10] 中身なんて特に

[11] しかもまたもや締め切りオーバー

タスクから見た場合

まずタスクがどのような状態をとりうるかを説明します。対応するマクロが include/linux/sched.h にあるので見てみましょう。

TASK_RUNNING
このタスクはすぐに実行することができます (現在 CPU で実行されている場合もあります)。

TASK_INTERRUPTIBLE

TASK_UNINTERRUPTIBLE
このタスクはすぐに実行することはできません。例えば，ディスクの IO 処理の完了を待っている等の状態です。ふたつの違いについては今回は触れません。

TASK_ZOMBIE
このタスクは _exit() システムコールにより終了しましたが，親プロセスがまだ終了コードを受け取っていません。

TASK_STOPPED
このタスクは SIGSTOP シグナルを受け取って停止しています。SIGCONT シグナルを受け取るまでは再開できません。

まず重要なことは，schedule() 関数を呼び出すことができるのはタスクが TASK_RUNNING であって，実際に CPU で実行されているときだけということです。

実際に schedule() を呼び出しているコードを見てみましょう。

```
asmlinkage int sys_pause(void)
{
        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return -ERESTARTNOHAND;
}
```

これは pause() システムコール[1] の実装ですが，schedule() 関数の典型的な使い方です。current 変数は現在のタスクを持っています。まず，現在のタスク (つまり自分自身) の状態を TASK_INTERRUPTIBLE に変更します。その後 schedule() 関数を呼ぶと，このタスクは TASK_RUNNING ではないので，絶対に実行されることはありません。そのかわりに他の適切なタスクが動き始めます。

このタスクがシグナルを受け取った場合，このタスクの状態は TASK_RUNNING に戻されます。他のタスクが schedule() を呼び出して，このタスクに CPU が回ってきた場合は，上のコードの schedule() 関数から帰ってきて，何事もなかったかのように残りのコードを実行します。

[1] pause() システムコールはシグナルを受け取るまで眠るというものです。

もうひとつコードを見てみましょう。

```
NORET_TYPE void do_exit(long code)
{
        struct task_struct *tsk = current;
        /* 中略 */
        tsk->exit_code = code;
        /* ここで current->state = TASK_ZOMBIE になる */
        exit_notify();
        schedule();
        BUG();
}
```

これは_exit() システムコールの実装です。終了コードを exit_code に保存した後，タスクの状態を TASK_ZOMBIE にして schedule() 関数を呼び出します。今回も TASK_RUNNING ではないので，このタスクに CPU が割り当てられることはありません。さらにこの後，親プロセスが終了コードを受け取ればこのタスクは消滅します。よって，今回の schedule() 関数は絶対戻ってきません (後に BUG(); というコードがありますが，これが実行されるとカーネルのバグということです)。

CPU から見た場合

CPU から見た場合，タスクは各レジスタの状態になります[2]。この状態のことをタスクのコンテキストといいます[3]。スケジューラが実行するタスクを入れ替える場合は，今動いているタスクのコンテキストを保存して，次に動かすタスクのコンテキストを復元すればいいわけです。

実際にスケジューラのコードを見てみましょう (kernel/sched.c にあります)。

```
/*
 *  'schedule()' is the scheduler function. It's a very simple and nice
 * scheduler: it's not perfect, but certainly works for most things.
 *
 * The goto is "interesting".
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */

asmlinkage void schedule(void)
{
      struct task_struct *prev, *next, *p;
```

[2]実際はもっと複雑ですが今は気にする必要はありません。
[3]すなわち，レジスタ以外のもっと複雑な部分もまとめてコンテキストといいます。

```
        /* 中略 */
        /*
         * This just switches the register state and the
         * stack.
         */
        switch_to(prev, next, prev);
        /* 中略 */
        return;
}
```

略している部分で次にどのタスクを選ぶべきか決めています。prev に前のタスクが，next に次のタスクが，それぞれ入った状態で switch_to() が呼び出されます。switch_to() はアーキテクチャによって異なるので詳しくは書きませんが，上に書いたように現在のコンテキストを prev->thread に保存し，next->thread に保存してあるコンテキストを復元します。

最後に，schedule() の頭にあるコメントは linux-0.0.1 からほとんど変わっていません (コードはかなり変わりましたが)。

次回は (あるんか)

今回はタスクが自分から CPU を空け渡していましたが，次回は他のタスクからどのように CPU を奪うかについて書く予定です。

DirectDraw の半透明処理第二回

まめ

はじめに

まめです。前回の DirectDraw による半透明処理の第二回です。今回も一応真面目です。締め切り 8 時間前ですが。

前回の訂正

前回の記事には確認された限りで二ヶ所ほど訂正するところがあります。一つ目。「アセンブラなのにインデントされている」などと書いてますが、編集長がインデント部分を削ってくれたのでインデントされていません。大したことじゃありませんが。

もう一つは、MMX による半透明処理部分で HASM さんの手により二つ命令が削られました[1]。残念ながら原稿の締切が迫っているので (覚えていたら) 次回紹介します。

これから出てくるアセンブラ部分はほとんど何も考えずに書いた記憶があり、見直す時間もないようなので、あらかじめ謝っておきます。ごめんにょ。

α 値任意の半透明処理

今までは α=50%でしたが、α が 50%以外で任意指定の関数を作ります。ShiftR、ShiftG、ShiftB、BitR、BitG、BitB はビデオカード依存の変数で、以下の通りです。

	ShiftR	ShiftG	ShiftB	BitR	BitG	BitB
RGB-565	11	5	0	0x1f	0x3f	0x1f
BGR-565	0	5	11	0x1f	0x3f	0x1f
RGB-555	10	5	0	0x1f	0x1f	0x1f

要するに Shift? は 16bit 中の？の色の部分へのスライド、Bit? は？の色の部分のマスクです。

```
VOID PutBlendAlpha(int x,int y,RECT rcSrc,int alpha)
{
 (中略)
  AddPitch1>>=1;
  AddPitch2>>=1;
  R1=ShiftR; R2=BitR;
  G1=ShiftG; G2=BitG;
  B1=ShiftB; B2=BitB;

  for (y=0;y<j;y++)
  {
    for (x=0;x<i;x++)
    {
      if ((Buff=data1[x])!=0x0000)
      {
        Buff2=data2[x];
        data2[x]=(((unsigned int)((((Buff>>R1)&R2)*alpha+((Buff2>>R1)&R2)*alpha2)>>8))<<R1)|
             (((unsigned int)((((Buff>>G1)&G2)*alpha+((Buff2>>G1)&G2)*alpha2)>>8))<<G1)|
             (((unsigned int)((((Buff>>B1)&B2)*alpha+((Buff2>>B1)&B2)*alpha2)>>8))<<B1);
      }
    }
    data1+=AddPitch1;
    data2+=AddPitch2;
  }

  lpDDSSystemSprite->Unlock(NULL);
  lpDDSSystemBack->Unlock(NULL);
}
return;
}
```

α 値を任意にした場合、

$$R\times\alpha+r\times(1-\alpha),\ G\times\alpha+g\times(1-\alpha),\ B\times\alpha+b\times(1-\alpha)$$

[1]予想はされていました (笑)

の式からしてかけ算が 6 回入ります。これだと 50000 回で 670ms かかりました。この式を

$$(R-r)\times\alpha+r,\ (G-g)\times\alpha+g,\ (B-b)\times\alpha+b$$

と変形すると、かけ算が半分になります[2]。この式を使った場合、50000 回で 650 ms でした。わずかばかりですが高速化しました。

これの MMX 化は面倒なんでやってません（^^;）

加算描画

今度は半透明処理ではなく、加算描画処理をしてみようと思います。

加算描画処理は、

$$\begin{aligned} R' &= R + r \\ G' &= G + g \\ B' &= B + b \\ if\quad R' > MaxR &\quad R' = \mathrm{Max}R \\ if\quad G' > MaxG &\quad G' = \mathrm{Max}G \\ if\quad B' > MaxB &\quad B' = \mathrm{Max}B \end{aligned}$$

として

$$R',\ G',\ B'$$

の色を描画する処理です。小難しく見えますが要するに色の三原色をそれぞれ足して、最大値を超えたら最大値にあわせる処理です (飽和加算)。

```
VOID PutBlendAdd(int x,int y,RECT rcSrc)
{
  (中略)
    AddPitch1>>=1;
    AddPitch2>>=1;
    R1=ShiftR; R2=BitR;
    G1=ShiftG; G2=BitG;
    B1=ShiftB; B2=BitB;

    for (y=0;y<rcSrc.bottom-rcSrc.top;y++)
    {
      for (x=0;x<rcSrc.right-rcSrc.left;x++)
      {
        if ((Buff=data1[x])!=0x0000)
        {
          Buff2=data2[x];
          if((R3=((Buff>>R1)&R2)+((Buff2>>R1)&R2))>R2) R3=R2;
          if((G3=((Buff>>G1)&G2)+((Buff2>>G1)&G2))>G2) G3=G2;
          if((B3=((Buff>>B1)&B2)+((Buff2>>B1)&B2))>B2) B3=B2;
          data2[x]=(R3<<R1)|(G3<<G1)|(B3<<B1);
        }
```

[2]やねうらお氏の記述を参考にしました。

```
        }
        data1+=AddPitch1;
        data2+=AddPitch2;
      }

      lpDDSSystemSprite->Unlock(NULL);
      lpDDSSystemBack->Unlock(NULL);
    }
    return;
  }
```

やはり条件分岐がつくと処理に時間がかかるため 50000 回に 1200ms ほどかかります。

MMX には paddusw という飽和加算のニーモニックがあるため、条件分岐が不要です。MMX を使用したコードを以下に載せます (これは RGB-565 または BGR-565 のビデオカード用のコードです)。

```
VOID PutBlendAddMMX(int x,int y,RECT rcSrc)
{
  (中略)
    _asm
    {
      mov   esi,data1   // 転送元
      mov   edi,data2   // 転送先
      mov   eax,0xf800f800
      movd  mm7,eax
      movq  mm0,mm7
      psllq mm0,32
      por   mm7,mm0     // 以上 RB マスク (0xf800f800f800f800)を作る処理\。
      mov   eax,0xfc00fc00
      movd  mm6,eax
      movq  mm0,mm6
      psllq mm0,32
      por   mm6,mm0     // 以上 G マスク (0xfc00fc00fc00fc00)を作る処理\。
      mov   eax,j
      mov   ebx,i
      mov   ecx,AddPitch1
      mov   edx,AddPitch2
      sal   ebx,3
      sub   ecx,ebx
      sub   edx,ebx
      sar   ebx,3
      sub   edi,esi
_y_loopstart_256_565:
       mov   ebx,i
_x_loopstart_256_565:
         movq  mm0,[esi]
         movq  mm1,[esi+edi]
         movq  mm2,mm0   // Src の処理
         pand  mm2,mm7   // R(B)のみ抽出
         movq  mm3,mm1   // Dest の処理
         pand  mm3,mm7   // R(B)のみ抽出
         paddusw mm2,mm3   // Src と Dest の飽和処理付き加算
         pand  mm2,mm7   // R(B)のみ抽出
         movq  mm3,mm0   // Gの処理 (R と同様)
         psllw mm3,5
         pand  mm3,mm6
         movq  mm4,mm1
         psllw mm4,5
         pand  mm4,mm6
         paddusw mm3,mm4
         pand  mm3,mm6
         psrlw mm3,5
         por   mm2,mm3   // R(B)と G の論理和
         movq  mm3,mm0   // B(R)の処理 (R と同様)
         psllw mm3,11
         pand  mm3,mm7
         movq  mm4,mm1
```

```
            psllw mm4,11
            pand  mm4,mm7
            paddusw mm3,mm4
            pand  mm3,mm7
            psrlw mm3,11
            por   mm2,mm3   // RGB の論理和
            movq  [esi+edi],mm2
            add   esi,8
_x_loopend_256_565:
            dec   ebx
            jnz   _x_loopstart_256_565
          add   esi,ecx
          dec   eax
          jnz   _y_loopstart_256_565
        emms
      }

      lpDDSSystemSprite->Unlock(NULL);
      lpDDSSystemBack->Unlock(NULL);
    }
    return;
  }
```

これで 50000 回で 210ms でした。およそ 6 倍のスピードアップです。

一応 MMX 部分の流れを説明します。

```
>movq   mm0,[esi]
>movq   mm1,[esi+edi]
```

読み込み。

```
>movq    mm2,mm0    // Src の処理
>pand    mm2,mm7    // R(B) のみ抽出
```

mm2 にスプライトをコピー。そこから mm7 は 0xf800f800f800f800 なので各ドットの R(BGR-565 の場合は B。以下 R のみ記述します) 部分のみを抽出します。

```
>movq    mm3,mm1    // Dest の処理
>pand    mm3,mm7    // R(B) のみ抽出
```

mm3 に背景部分をコピー。そこから各ドットの R 部分のみを抽出します。

```
>paddusw mm2,mm3    // Src と Dest の飽和処理付き加算
```

mm2 と mm3 を飽和加算します。もし和がオーバーフローしたパックが出た場合、mm2 のオーバーフローしたパックに 0xffff が代入されます。

```
例)paddusw mm0,mm1
  mm0 : 1000000000000000 0000000000000000 0100000000000000 1111111111111111
  mm1 : 1000000000000000 0000000000000001 0100000000000000 1111111111111111
 結果          飽和            非飽和           非飽和            飽和
  mm0 : 1111111111111111 0000000000000001 1000000000000000 1111111111111111
```

```
>pand    mm2,mm7    // R(B) のみ抽出
```

オーバーフローしたドットが出た場合、そこは 0xffffffff の対策です。

```
>movq    mm3,mm0    // G の処理
```

以下 G について R と同様の処理をします。

```
>psllw    mm3,5
```

スプライトの G の部分を左に 5 つシフトします。この位置にシフトすることで、paddusw の加算で飽和したときに全ビットがたつようにします。

```
>pand   mm3,mm6
```

スプライトの G の部分のみ抽出します。mm6 は 0xfc00fc00fc00fc00 です。

```
>movq   mm4,mm1
>psllw    mm4,5
>pand   mm4,mm6
```

同様に背景部分の G の部分のみ抽出します。

```
>paddusw  mm3,mm4
```

飽和加算します。

```
>pand   mm3,mm6
```

オーバーフロー対策。

```
>psrlw    mm3,5
```

もとの G の位置に戻す。

```
>por    mm2,mm3   // R(B) と G の論理和
```

R と G の部分の論理和を取ります。

```
>movq     mm3,mm0   // B(R) の処理
>psllw    mm3,11
>pand     mm3,mm7
>movq     mm4,mm1
>psllw    mm4,11
>pand     mm4,mm7
>paddusw  mm3,mm4
>pand     mm3,mm7
>psrlw    mm3,11
```

R や G と同様です。

```
>por    mm2,mm3   // RGB の論理和
```

RGB すべての論理和をとります。

```
>movq   [edi],mm2
```

背景部分にスプライトを描画します。

拡大縮小

今までの処理はすべて等倍でした。今度は拡大処理もある半透明処理をやってみようと思います。縮小処理は実用性が少ないように思うので対応しません（^^;）半透明処理は 50% 限定です。透過処理、クリッピングを行なっています。

```
VOID CAyu::PutBlendSize(RECT rcDest,RECT rcSrc)
{
  static  DDSURFACEDESC2 ddsd1,ddsd2;
  static  LPWORD data1,data2,p1,p2;
  static  long AddPitch1,AddPitch2;
  static  UINT Buff,Buff2;
  static  UINT x,y,Ex,Ey;
  unsigned i,j,nsx,nsy,ndx,ndy;

  if(AccessMem) return;

  nsx=rcSrc.right-rcSrc.left;
  nsy=rcSrc.bottom-rcSrc.top;
  ndx=rcDest.right-rcDest.left;
  ndy=rcDest.bottom-rcDest.top;

  if(rcDest.left  <0  ) rcSrc.left  +=   -rcDest.left*nsx/ndx   ,rcDest.left  =0;
  if(rcDest.top <0  ) rcSrc.top   +=   -rcDest.top*nsy/ndy    ,rcDest.top   =0;
  if(rcDest.right >=sx) rcSrc.right -=(rcDest.right-sx)*nsx/ndx ,rcDest.right =sx;
  if(rcDest.bottom>=sy) rcSrc.bottom-=(rcDest.bottom-sy)*nsy/ndy  ,rcDest.bottom=sy;

  if(rcSrc.bottom-rcSrc.top>0 && rcSrc.right-rcSrc.left>0)
  {
    ZeroMemory(&ddsd1,sizeof(DDSURFACEDESC)); ddsd1.dwSize=sizeof(DDSURFACEDESC);
    ZeroMemory(&ddsd2,sizeof(DDSURFACEDESC)); ddsd2.dwSize=sizeof(DDSURFACEDESC);

    if(lpDDSSystemSprite->Lock(NULL,&ddsd1,DDLOCK_WAIT,NULL)!=DD_OK) return;

    if(lpDDSSystemBack->Lock(NULL,&ddsd2,DDLOCK_WAIT,NULL)!=DD_OK)
    {
      lpDDSSystemSprite->Unlock(NULL);
      return;
    }

    AddPitch1=ddsd1.lPitch;
    AddPitch2=ddsd2.lPitch;
    data1=(LPWORD)ddsd1.lpSurface+rcSrc .left+rcSrc .top*(AddPitch1>>1);
    data2=(LPWORD)ddsd2.lpSurface+rcDest.left+rcDest.top*(AddPitch2>>1);

    i=(unsigned)(rcDest.right-rcDest.left)>>2;
    j=(unsigned)(rcDest.bottom-rcDest.top);

    nsx=rcSrc.right-rcSrc.left;
    nsy=rcSrc.bottom-rcSrc.top;
    ndx=rcDest.right-rcDest.left;
    ndy=rcDest.bottom-rcDest.top;

    Buff=MaskRGB;
    if(UseAsm)
    {
      _asm
      {
        mov   esi,data1         // 転送元
        mov   edi,data2         // 転送先
        mov   eax,Buff
        mov   ebx,eax
        shl   eax,16
        or    eax,ebx
        movd  mm4,eax
```

```
        movq  mm0,mm4
        psllq mm0,32
        por   mm4,mm0           // 以上 RGB マスクを作る処理。
        mov   eax,0x0000ffff
        pxor  mm7,mm7
        movd  mm7,eax
        mov   eax,j
        mov   ebx,i
        mov   ecx,AddPitch1
        mov   edx,AddPitch2
        sal   ebx,3
        sub   edx,ebx
        sar   ebx,3
        mov   ebx,0
_y_loopstart:
          push  esi
          push  eax
          push  ebx
          push  ecx
          push  edx
          mov   eax,0
          mov   ebx,nsx
          mov   ecx,ndx
          mov   edx,i
_x_loopstart:
            movq  mm0,[esi]     // 転送元
            movq  mm1,mm0
            pand  mm1,mm7       // １ドット目を抽出
            psllq mm7,16        // マスクを２ドット目にあわせておく
            add   eax,ebx
            cmp   eax,ecx
            jl    _j1a
            sub   eax,ecx
            add   esi,2
            jmp   _j1b
_j1a:         psllq mm0,16        // ドットを進めない
_j1b:         movq  mm2,mm0
            pand  mm2,mm7       // ２ドット目
            por   mm1,mm2
            psllq mm7,16
            add   eax,ebx
            cmp   eax,ecx
            jl    _j2a
            sub   eax,ecx
            add   esi,2
            jmp   _j2b
_j2a:         psllq mm0,16
_j2b:         movq  mm2,mm0
            pand  mm2,mm7       // ３ドット目
            por   mm1,mm2
            psllq mm7,16
            add   eax,ebx
            cmp   eax,ecx
            jl    _j3a
            sub   eax,ecx
            add   esi,2
            jmp   _j3b
_j3a:         psllq mm0,16
_j3b:         movq  mm2,mm0
            pand  mm2,mm7       // ４ドット目
            por   mm1,mm2
            psrlq mm7,48
            add   eax,ebx
            cmp   eax,ecx
            jl    _j4a
            sub   eax,ecx
            add   esi,2
_j4a:
            movq  mm0,[edi]   // mm1 に背景部分４ドット分読込
            pxor  mm3,mm3     // mm3 をクリア
```

```
              pcmpeqw mm3,mm1    // マスクを作成
              psrlw mm1,1        // mm0 を１つ右シフト（２で割る）
              pand  mm1,mm4      // mm0 を MaskRGB でマスクする
              movq  mm2,mm0      // mm1 を mm2 にコピー
              pand  mm0,mm3      // mm1 は 透過する部分のみ残す
              pandn mm3,mm2      // mm3 は 透過しない部分のみ残す
              psrlw mm3,1        // mm3 を１つ右シフト（２で割る）
              pand  mm3,mm4      // mm3 を MaskRGB でマスクする
              por   mm0,mm3      // 透過する部分としない部分の論理和
              paddw mm0,mm1      // スプライトと背景の和を取る
              movq  [edi],mm0    // 背景部分にスプライトを書き込む
              add   edi,8
              dec   edx
              jnz   _x_loopstart
            pop   edx
            pop   ecx
            pop   ebx
            pop   eax
            pop   esi
            add   ebx,nsy
            cmp   ebx,ndy
            jl    _jy
            sub   ebx,ndy
            add   esi,ecx
_jy:          add   edi,edx
            dec   eax
            jnz   _y_loopstart
          emms
        }
      }
      else
      {
        i<<=2;
        AddPitch1>>=1;
        AddPitch2>>=1;
        Buff2=MaskRGB;
        for (y=0;y<j;y++)
        {
          for (x=0;x<i;x++)
            if(Buff=data1[(x*nsx)/ndx])
          {
            data2[x]=((Buff>>1)&Buff2)+((data2[x]>>1)&Buff2);
          }
          if((y*nsy)/ndy<((y+1)*nsy)/ndy) data1+=AddPitch1;
          data2+=AddPitch2;
        }
      }
      lpDDSSystemSprite->Unlock(NULL);
      lpDDSSystemBack->Unlock(NULL);
    }
    return;
}\
```

MMX 部分の拡大処理部分は、ブレゼンハムのアルゴリズムみたいなアルゴリズム (正しくはなんというのかわかりません) を使用しています。

```
>pand mm1,mm7        // 1 ドット目を抽出
>psllq  mm7,16       // マスクを 2 ドット目にあわせておく
>add  eax,ebx
>cmp  eax,ecx
>jl   _j1a
>sub  eax,ecx
>add  esi,2
>jmp  _j1b
>_j1a:
```

```
>psllq  mm0,16       // ドットを進めない
>_j1b:
```

1 ドットあたりに行なう処理です。eax に ebx を足して、ecx より大きくなったら eax から ecx を引いているあたりがブレゼンハムっぽいです (笑)。eax が ecx より小さかったら次のドットには進まないように、Src を 1 ドット分左シフトしています。

このようにすることで、Dest に書き込む分と同じだけ Src を読みとることになります。もし一つ一つのドットに対してかけ算やわり算を行なった場合、Src を MMX レジスタ分 (すなわち 4 ドット分) 読みとることになります。高速化のためには、出来るだけメモリアクセスの機会を削ることが重要なので、このようにややこしい処理にしています。

次回予告

もともとこの原稿は長いひとつの原稿でしたが、長すぎるということで全 3 回になりました。が、その原稿の内容は前回と今回で使い切ってしまいました。次回はブレゼンハムの直線アルゴリズムを使用して、直線の半透明をやってみようと思います。[3]

2001 War in Mountain

もりさわゆうな

なぜ山に登るんですか?

—そこに山があるからです。

そうです、だから人は山に登るんです。というわけで、ボクも山に登ってきました。とはいえ、普通の山ではなく、名古屋にある「軽食喫茶　マウンテン」(以下マウンテン) というお店に登ってきました。[1]

マウンテンが人々の心を引きつけるのは、その斬新かつ新機軸なメニューである、と言い切ることが出来ます。その「料理」を感触することが出来た人間のことを「登頂成功」といい、英雄の称号を人々から送られます。

そのメニューとはいったいなんでしょうか。メニューの中で一番有名な「甘口スパゲティー」をみていていただきましょう。

[3] ブレゼンハムのアルゴリズムはかなり古い部報にあったような気がするので、やっぱり違う内容になるかも知れません。つまり未定です。

[1] 編注:以下の文章には大量のアニメねた、その他もろもろが含まれております。分らない方は分らないままのほうが幸せかもしれません。

甘口イチゴスパ	800 円
甘口抹茶小倉スパ	800 円
甘口バナナスパ	800 円
バナナスパ	800 円
メロン風スパ	800 円

……。

いかがでしょう。この名前をみるだけで我々の中の冒険心を刺激されるメニュー!! このメニューの実物をみたい、それだけで東京から名古屋までいく理由は十分です。名古屋にいるボクの相棒の神楽坂祐里くんと、連絡をとり早速旅立ちました。本当はその後に「さゆけっと」とかいう Kanon-倉田佐祐理&川澄舞 Only 同人即売会に参加する予定だったのですが、そんなことは二の次。このところ不景気で、同人即売会の売り上げがいまいちだったりするので、交通運賃さえでてくれれば問題なし、とおもっていました。

マウンテン登頂予定日前日、所用により小松から名古屋いりし、そのまま神楽坂邸に収容。名古屋人ののえる君とかわぐつ君を Hook することに成功したわたしは、翌日にそなえてぐっすり寝ました。

当日、朝 6:30 頃、神楽坂邸を出発し、一途マウンテンまで向かいました。地下鉄を二回ほど乗り換え、マウンテンの最寄り駅である舞鶴線「いりなか駅」に到着。とちゅう「川名」という駅があり、みさき先輩がわれわれの登頂を応援してくれてると灌漑にひたっていたのはいうまでもありません。とにかく、無事いりなか駅で、のえる君、かわぐつ君と合流。そして歩いてマウンテンへの道を一歩一歩進みます。

そして、歩くこと数十分、マウンテン.mpeg や、WebSite でみてきた「あの」マウンテンは目の前にみえました。古い洋風の建物で、ちょっとお世辞にも周りとマッチングしてない建物……。この恐怖の牙城にわれわれは一丸となり、進軍しました。

中は予想通り薄暗く、危険なかおりをほのめかしています。[2] 我々は左奥の窓のそばの席につきました。着座して、まず驚愕の事実を発覚しました。席が４人掛けのくせにテーブルかなり小さいということです。ここから苦難があろうとは……。するとのえる君が一言、

「小さいテーブルで皿をささえながら食べるんだよ」

教えてくれました。そうか、ここは名古屋なんだ、と自分にいい聞かせ、兎にも角にも注文をしました。

—甘口イチゴスパ。

[2]Tonight2 とかでやってたんですが、あんなに店内あかるくねーよ、ともってました。

と私は果敢にももっとも危険な牙城の攻略を試みました。たとえれば、爆牌リーチにたいして、そのまま危険牌をつっこむようなものです。[3] 他の方々は日頃これるみたいなので、モーニングセットとか、焼き肉トーストとかたのんでました。奥にいたマスターと奥さんがにやりとしてたのは言うこともありません。こんな朝 8 時から行くのは危険だという証でしょう。しかし漢だったら敢えて地雷を踏む勇気も必要である。この遂行な理想のために……ジーク・ジオン![4]

待つこと数分、ちょっと普通よりおおきい皿に、ピンク色の麺とピンク色のイチゴらしき物体ののったスパゲティーが到着しました。手元にあった、フォークを手にとり、誇り高きピンク色の戦士[5]との戦いを決意しました。

とりあえず一口。

……。

この文章を書いてる今でもあの感触は一生忘れられません。黒騎士[6]と対峙した、アトール一般兵みたいなもんでしょう。例をかえれば、ノワールが強敵だとわかった、ザコ敵みたいなもんでしょう。[7] あたためた苺が喉を通る感触の「あの」気持ちわるさは体験したものしかわかりません。しかし、もう一口すすめなければ完食への道はとざされてしまいます。「我は放つ光の牙！」[8]とかいいつつ、二口目、三口目とすすめていったが、ついに五口目に私に神が降臨してきました。

—喉の食べ物を通すことが体より、脳が拒否している……。

そう神はわたしに告げると、私の意識はだんだん遠くなってきました。しかし、そこはミラージュ騎士たるもの、なんとか意識を復活させ、奥さんがをついでくれた水を口に含みました。そして荷物を整理し、お会計をすませると、あわててタクシーに乗り込みました。そう、時間がなかったのです。サークル入場まできりぎりの時間でした。かわぐつ君とのえる君とはお店の前で別れ我々は一途さゆけっと会場へと向かいました。

[3] ノーマーク爆牌党（片山まさゆき）より。打つと煙がでてきたりする。これを再現するためにタバコのけむりでやろうとした者もいたみたいだが、焦げてガンパイになり、雀荘から出入り禁止を食らったというのは記憶に新しい。

[4] これを書いた当時、DVD で 0083-スターダストメモリーをみていました。祝デンドロニウムプラモ化。

[5] MS ではなく FiveStarStories の MH です。マイティーシリーズの暁姫とか。ピンク色というと、「アイシャ様!」というのが記憶に新しいです。

[6] 黒騎士は、L-GAIM のバッシュではなくて、FSS のバッシュ・ザ・ブラックナイトです。想定してるのは、アトールの聖都にバッハトマが降りてくるというところでしょうか。頼むから連載続けてくれー。はあはあ。あ、ヤーン・バッシュ王女でもないですので、あしからず。

[7] ノーワルの夕霧のガレキはいつ完成するんでしょうか。夕霧たまりませんね、ええ。

[8] オーフェンの呪文ってこれであってたっけ?

で、転んでもただで起きない私は、タクシーの中で気合いで Kanon の SS を書いてました。一応ここから掲載しますが、載せるかどうかは編集部のご判断におまかせします。[9]

とにもかくにも無事イベントで交通費を稼いだ私は、そのまま名古屋からグリーン車でまったり帰りました。次は夏コミだなぁ、と考えていました。[10]

今回の結論:「マウンテンに行くのは乗り換えが大変である。」

以下 SS。

「あはは……。これなんでしょうかね。」
薄暗い建物の中、佐祐理と舞は運ばれてきた料理をみて目がまさしく点になっていた。

明かに机に対して大きすぎる皿の上には、赤色の麺と紅白に彩られたいちごとクリームが乗っかっていた。その名もイチゴクリームスパゲティー。

舞と佐祐理はたまたま通りかかった喫茶店の入ると、メニューの中から前述のスパゲティーを頼んだ。理由は簡単だ。

「おやつと食事が一緒にできるから。」

「うっ……たべようか……。」
舞はこくん、とうなずくと、ナプキンに包んであったフォークを手に取りその麺をくるくる巻き始めた。明かにクリームがついていない部分をとり口へと運んだ。
「……。」
舞はだまってまたその赤色の麺を口に運んだ。

「舞……大丈夫？」
またこくん、とうなずくと佐祐理もスプーンをとって巻きだした。

舞は二口、三口とすすみ、四口目といったとき、急に違和感を感じた。
「うっ……。」
口の中にはイチゴ色の麺の中にすりつぶしてあるだろうと思われるイチゴの果実の残りが舞の歯にからみついた。その果実の残りの麺を食道に通すことには不快、いや舞の本能が拒絶していた。

[9]編注:掲載することにしました。
[10]三日目の東 1D51-b「Twinkle Nightz」です。よろしくおねがいしまーす。

それでも舞は無理矢理、食道に通していく。

麺が半分に達した頃に、舞には次の難題が降りかかってきた。麺のそばにおいてある生クリームである。その生クリームはスパゲティーという比較的モダンな食べ物には到底マッチするはずがない。舞は机のそばにおいてあった水を口の中に含むと、その生クリームを一気に口の中に押し込む。
「ぐぁぁ……。」
水と生クリームの関係はまさに水と油。口の中は余計混乱をきたし舞はパニック状態に落ち込む。

すでに人としての食事を行うという営みを拒絶しかかってる舞。そのスプーンを口にくわえ、両手がぶらりとだらしなく垂れ下がる。

ふと隣のほうをみると佐祐理がにっこりとこっちを観ている?「舞……それもらっていい？」
視線を下のほうに落とすと同じモノをたべてる佐祐理の皿にはクリームかけら一つとしてなかった。

編集後記

★ ただ今、7 月 6 日午前 0 時 30 分。編集終了。

★ むむむぅ…、あまりに平和だ。平和すぎる…。なんの滞りもなく編集が終わってしまった。何故だ。

★ なにか裏があるのだろうか。例えば、学生会館のゲスプリが全部故障するとか…。それとも……

★ ……はっ、縁起でもないことを考えていた。

理論科学グループ　部報 第 236 号

2001 年 7 月 6 日　発行

発行者　西田健志

編集者　會田雄亮

発行所　理論科学グループ

〒 153–0041　東京都目黒区駒場 3–8–1

東京大学教養学部内学生会館 305

Telephone: 03–5454–4343

理論科学グループ部報 第 236 号
— 新入生自己紹介号 —
2001 年 7 月 6 日

THEORETICAL SCIENCE GROUP