計算機科学'11 ＃5 — マルチメディアとWeb上の描画

久野　靖*

2011.6.21

1　アナログとデジタル再訪

1.1　デジタル化とマルチメディア

アナログとは値が連続的に変化するもの、デジタルとは値が有限個の場合のうちの1つという形で表されるもの、という説明は既にしたと思います。では改めて考えてみて、我々の身の回りにある情報は、アナログとデジタルのどちらが多いでしょうか?

答えは「アナログ情報」であるはずです。私達が見たり聞いたり感じたりするものの大きさ、形状、色合い、音色や音の強さ、重さ、手ざわり、他人の表情や見ぶりなどは、すべてアナログ情報なのですから。

ただし、文字で表された情報だけはデジタル情報です。なぜなら文字とは、かなであれば50通り、漢字を含めても数千通りの場合のうちどれであるかを表すものであり、たとえば「あ」という文字がここに書かれていたとして、それが「あ」と読みとれる限り、どんな形や色であるかということは問題にされないからです。[1]

デジタル情報には、コピー、保管、伝送などにおいて劣化しにくいという特徴があります。たとえばコピー機などない昔においては、情報をコピーするのには手で写し取るしかありませんでしたが、文字で書かれた情報 (デジタル情報) であれば文字を書き間違えない限りは写し取ったものも文書としては同じ内容なわけですし、何回書き写しても同様です。これが絵などのアナログ情報では、いかにうまく模写してもオリジナルとは微妙に違うわけですし、模写の模写の模写は本物とはかなり違ってしまうことでしょう。

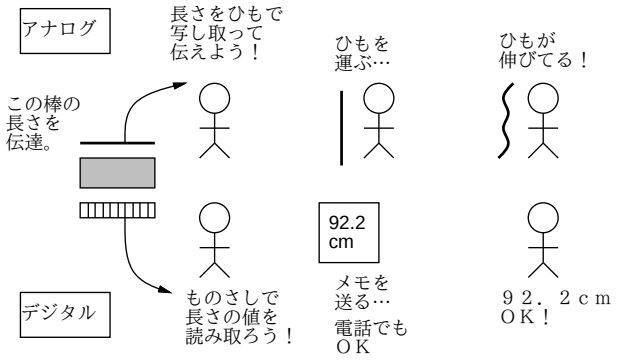

図 1: アナログとディジタルによる伝達

*経営システム科学専攻

[1]ただし、書体や色などのデザインを鑑賞する場合はやっぱりアナログ情報と言えるでしょう。言葉を聞くのは? 言葉ではその抑揚や調子が重要な情報なので、完全にディジタルとは言いにくいですね。

保管についても、年を経た古文書であっても、文書ならば文字が何であるか読みとれる限りにおいては同じ文書が復元できますが、色褪せた絵の場合は元がどんな色だったかは推測するしかありません。伝送については、たとえば「手元の鉛筆の長さ」を電話で相手に伝えることを考えてください。数字を使ってよければ (つまりデジタル情報にしてよければ)、物差しで計ってその数字を読み上げれば簡単です。しかし、もし数字を使わないとすれば、何か手近なものになぞらえる等の方法で伝達したとしても、正確というのは難しいはずです (図 1)。

今日デジタル情報がもてはやされているもう 1 つの理由として、計算機で扱えることが挙げられます。従来のアナログ情報であれば、画像ならカメラ、音ならテープレコーダーというふうに、記録や保管にはそれぞれ専用の機器が必要でした。しかし計算機は任意のデジタル情報を扱う装置ですから、デジタル化して計算機に入れればどのような情報でもまとめて扱えます。一般に、従来からのデジタル情報である文字情報に加えて、画像、動画、音声など複数の種類の情報を一緒に扱うことを**マルチメディア**と呼びます。そして、計算機は単にこれらの情報を記録/再生するだけではなく、プログラムを使ってさまざまに加工したり、ネットワーク経由であちこちに送ったりもできるわけです。計算機は見方を変えれば、マルチメディアのための土台 (プラットフォーム) だとも言えるわけです。

1.2 AD 変換と DA 変換

画像や音など、もとがアナログの情報であれば、これらを計算機で扱うためにはデジタル情報に直さなければなりません。これを **AD 変換** (アナログ=デジタル変換) といいます。そして、これらの情報を人間が鑑賞 (?) するためには、元の形すなわちアナログ情報に戻す必要があります。これを **DA 変換** (デジタル=アナログ変換) といいます。そして AD/DA 変換にはそれ固有の限界があることには注意が必要です。

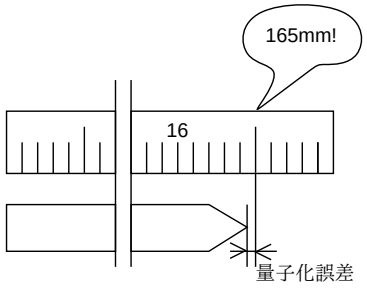

図 2: 量子化誤差

たとえば、先の「電話で鉛筆の長さを伝える」例を再考してみましょう。

鉛筆の長さをものさしで計って「165mm」と読み上げたとします。しかし、実際に鉛筆の長さがちょうどミリ単位でぴったり、ということはないでしょう。ですから、実際の長さは 164.5mm〜165.5mm の間のどこかあたり、というくらいしか分かりません。ではノギスで 0.1mm まで計ったら? またはマイクロメータで 0.01mm まで計ったら? そのようにしていくら「単位」を細かくしても、その単位いくつ分、という形で表す以上、「単位未満」の部分は数値として現れて来ないで無視され、得られた情報が「階段状」になることに変わりはありません。これを**量子化誤差** と呼びます。AD 変換において、量子化誤差は本質的に避けられないものです (図 2)。

また、「いつ」計測を行うかという問題もあります。たとえば音のように連続的に変化する信号をデジタル化する場合は、図 3 左のように、連続的に変化するアナログ値に対して一定時間ごとに「測定」を行って値を取得することになります。この「測定する時の値」のことを**サンプル**値と言います。とびとびのサンプル値の連なりで変化の様子を表すわけですから、サンプル値を取得する時点と時点の間のようすは無視され記録されません。加えて、そのサンプル値をディジタル値として読み取るため、量子化誤差が加わります。

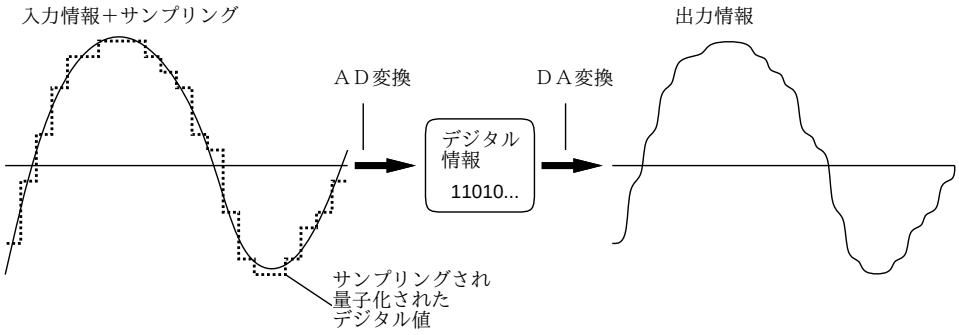

図 3: AD 変換と DA 変換

逆に DA 変換においては、「サンプル間隔ごとの」「単位いくつ分で表された」ディジタル情報から元のアナログ情報を復元するので、その「階段」のすき間は適当に埋めることになりますが、図 3 右のように「ギザギザ」が残ってノイズとして知覚されたりすることもあります (デジタル処理された画像などでよく見掛けます)。もちろん、「きざみ」を十分細かくすることで元の情報により忠実にすることはできるのですが、無限に細かくすることはできません。ともかく、このギザギザはサンプリングと量子化による誤差、つまりディジタル化したことにもともとの原因があるわけです。

2 サウンド

2.1 サンプル形式のサウンド

マルチメディア情報の具体例として、まずは**音** (**サウンド**) を取り上げることにしましょう。そもそも音とは何でしょうか? 小学校の理科で習ったはずですが、音とは「空気の振動」であり、空気の圧力の変化が連続的に伝わって行くものです。したがって、音の情報を記録しようと思えば、たとえば空気の圧力を電圧などに変換して (これは**マイクロフォン**が行うお仕事)、その電圧の変化を何らかの形で記録すればよいわけです。逆に、その電圧の変化を空気の圧力に変換すれば、同じ音になって聞こえます (これはもちろん**スピーカー**のお仕事です)。

ここまでは空気の圧力に比例する信号、すなわちアナログ信号を考えていましたが、計算機で扱うためにはこれをディジタル化する必要があります。その原理については前節で説明しましたが、用語定義も含めて改めて整理して示すと次のようになります:

- アナログ信号から、非常に短い一定の時間間隔でサンプルを採取する。
- 各サンプルは通常、8 ビットないし 16 ビットの 2 進数として記録する。

この処理を **PCM**(パルス符号変調)、このような形で記録/表現されている音のことを、**サンプル形式**のサウンドと呼びます。ここで、

サンプルを取得する間隔のことを**サンプリングレート** (ないし**サンプリング周波数**) と言います。たとえば音楽用 CD は 44KHz(1 秒間に 44,000 回) のサンプリングレートを用いています。[2]実際にどれくらいのサンプリングレートやビット数が可能かどうか、またその「音質」がどうかということは、計算機に接続されている**サウンドカード** によって決まってきます。今日のできあいの PC などでは、最初からそれなりの音質のサウンドカードが付属していることが多いでしょう。

なお、サンプルのビット数が 8 とか 16 なのは、計算機の方でデータがバイト単位だと扱いやすいためです (12 ビットを使う場合もあります)。ここで 8 ビットなら 256 段階、16 ビットなら 65,536 段

[2]理論的には、XKHz の音を記録するには $2X$KHz のサンプリング周波数があれば十分です。人間の耳に聞こえるもっとも高い音は 20KHz と言われているので、それよりやや高い 22KHz まで記録するために 44KHz というサンプリングレートが選ばれたそうです。

階の信号の強さが識別できるので、当然ながら 16 ビットの方が高音質になります (CD も 16 ビットを採用しています)。そして、ステレオであれば左右チャンネルごとにサンプルを採取します。

たとえば、5 分間 (300 秒) の音声を 44KHz、16 ビットステレオでディジタル化するとどれくらいのデータ量になるでしょうか?

$$300 \times 44,000 \times 16 \times 2 = 422,400,000 bits$$

ですから、バイト (8 ビット) 単位に直すと 53 メガバイトくらいになります。ちなみに 6 百万ピクセル (1 ピクセル 24 ビット) のデジカメ画像だとそのまま保存しても 144,000,000 ビットなので音の方がずっと多いですね。

サンプリングレートやビット数を減らせばこれを少なくできますが、当然ながら音質は悪くなります。

整理すると、サンプル形式サウンドの場合は、その原理から「きめ細かさ」を次のパラメタによって調節できます:

- サンプリングレート (8KHz〜64KHz くらい)
- サンプルのビット数 (8 ビット、12 ビット、16 ビット)
- リニア、ノンリニア (目盛りのつけ方を対数的にする)
- チャネル数 (モノラル/ステレオ)

以上が基本的なパラメタですが、これを格納するファイル形式として代表的なものに次のものがあります:

- 生の PCM 形式 — 上記のデータを「そのまま」記録したもの
- **WAV** 形式 — 同上だが、ファイルの先頭にサンプリングレート等の情報を記録した部分がついている
- **QuickTime** 形式 — WAV の Apple 版と思えばよい
- **MP3** 形式 — MPEG Layer 3 が正式名称。圧縮を行うことで、生のデータにくらべて大きさが 10 分の 1 くらいにできる。

とくに最後の MP3 形式は、これを使えば CD1 枚が 50 メガバイトくらいにできるため、音楽データの配布に広く使われるようになりました。

これは便利なことではありますが、その反面、著作権を無視した音楽データの配布にもつながっていて、その影響による CD の売上減少は音楽産業やそこから収入を得ているアーティストにとって看過できない状態であるとも言われています。このため、著作権管理のための制御機能を追加したファイル形式も提案され使われるようになっています。このような、ディジタル情報の著作権管理のことを一般に **DRM**(Digital Rights Management) と呼び、ホットな話題の 1 つです。

たとえば、自分が持っている CCCD(コピーコントロール CD) の曲がオーディオプレーヤに入れられなくてイライラしたことはありませんか? 「自分が持っている」のだから本来コピーできてよいはずですよね。また、ネット経由で音楽を購入する場合なども DRM つきの形式になっていて、自分の PC やプレーヤでしか聴けないなどの制約が掛かることが普通です。

あと 1 つ、ここまではすべて「サンプル形式のサウンド」についての説明でしたが、音を記録する方式にはもう 1 つ **MIDI** 形式があります。これは「どんな楽器の」「どんな高さの音を」「いつ」「どれくらいの強さで」「どれくらいの長さ」出すか、という情報を記録する方式で、一見面倒そうに見えますが、音の波を記録するサンプル形式よりずっとコンパクトになりますし、後から (加工ではなく) 編集できます。つまり、楽譜の情報を持っているようなものなので、その楽譜を手直しできるわけです。

2.2 サンプル形式サウンドの生成

お話ばかりでは面白くないので、音の話の締めくくりとして、簡単な C プログラムで音を合成して聴いてみることにしましょう。ここではプログラムからは 16 ビット 2 チャンネル 44KHz のデータを生のまま (PCM 形式) で送り出し、これを lame というプログラムで MP3 形式に変換し、続いて madplay という MP3 再生プログラムで再生して聴くことにします。

とりあえず生成プログラムを示しましょう。配列 a は要素数 2、各要素が 16 ビットの配列で、0 番目が左チャネル、1 番目が右チャネルの音の出力値を保持するのに使います:

```
/* gensound.c --- generate sound wave */
#include <math.h>
#include <stdio.h>
#define SEC 44000
#define D   1.05946309
double t[100];
unsigned short a[2];

/* n --- 配列 t の何番目を使うか。1 つの「音」毎に別の番号に! (0〜99)
   hz --- 何 Hz の音を出すか。440 が「ラ」。
   vol --- 大きさ、全部の音の vol を合計して最大 15000 に収めること!
   pos --- 左右のバランス、0.5 が中央。 */
wave(int n, double hz, double vol, double pos) {
  double x = sin(t[n] += 2*3.1416*hz/SEC);
  a[0] += (int)(vol*pos*x); a[1] += (int)(vol*(1-pos)*x);
}

main() {
  int i;
  for(i = 0; i < 10*SEC; ++i) { /* 10 秒間 */
    a[0] = a[1] = 15000;        /* 出力配列の初期化 */
    wave(0, 440, 1000, 0.5);
    if(i > 2*SEC) wave(1, 440*D*D*D*D, 1000, 0.2);
    if(i > 4*SEC) wave(2, 440*D*D*D*D*D*D*D, 1200, 0.8);
    if(i > 6*SEC) wave(3, 880, 800, 0.5);
    fwrite(a, sizeof(a), 1, stdout);  /* 出力 */
  }
}
```

このプログラムは sin 曲線を持つ音 (正弦波) を同時に 100 音まで重ねて出力できます。それぞれの波は

$$l = v \times p \times sin\theta, r = v \times (1-p) \times sin\theta$$

により表されます。l、r はステレオの左と右の音量、v は音の大きさ、p は音の左右のバランス (0.5 だと左右が同じ音量) を表します。

ここで角度 θ(単位はラジアンとする) は時間の関数であり、たとえば 440Hz の音 (中央のラ) であれば $\frac{1}{440}$ 秒につき 2π の割合で変化させる必要があります。このプログラムでは変数 i を 0、1、…と変化させながら、各時点での数値を出力して行きます。i が 1 進む間の時間を 1tick(「きざみ」の意味) と呼ぶことにすると、サンプリング周波数を 44KHz とした場合、44,000 は 440 の 100 倍なので θ の変化は 100tick につき 2π、1tick につき $\frac{2\pi}{100}$ ということになります。

ラの音はそれでいいとして、それ以外の音はどうすればいいのでしょうか? 実は (といっても、ご存じの方が多いと思いますが)、音というのは周波数が 2 倍になると 1 オクターブあがって聴こえます。ですから 440 に 2 を掛ければ、1 オクターブ上のラになります。でもその途中の音は…? それは、1 オクターブの間は半音が 12 個に分けられ、その 1 半音ごとに同じ比率で周波数が上がっていくわけですから、[3]

$$D \times D \times D \times D \times D \times D \times D \times D \times D \times D \times D \times D = 2$$

になるような D を計算してやれば、440 に D を 1 回掛ければ半音あがってラ♯、2 回掛ければ 1 音上がってシの音が作れます。実際、そのような D を計算してこのプログラムの冒頭に定義してあります。[4]

さてプログラムに戻って、1 つの波につき配列 `t` の 1 つの箱を使って上記 θ を覚えておき、音の周波数に応じて上の公式を使って θ を増加させて行くことで複数の波を並行して発生させます。

関数 `wave` は箱の番号、周波数、音量、バランスを受け取って左右チャネルの波の大きさを計算し、`a[0]` と `a[1]` に足し込みます (なぜ足し込むかというと、複数の波つまり音を重ね合わせるには足して行く必要があるからです)。

次に `main` を見てください。ここでは変数 `i` を 0 から 440,000 まで順に増やしながら (つまり 440,000tick で 10 秒間)、各 tick ごとの音を生成しています。各 tick ごとに `a[0]` と `a[1]` は 15000(中くらいの値) で初期化し、これに `wave` を使って音の値を重ね合わせていきます。まず、常に 440Hz の音を出し、2 秒後からはその 3 度上、4 秒後からは 5 度上、6 秒後からはオクターブ上の音を増やすことで和音をつくり出しています。もちろん、ある秒からある秒までの間だけ音を出したければ

```
if(i > 2*SEC && i < 3*SEC) wave(...)
```

のようにすればよいわけですが。

最後に、このプログラムの音を聞く手順を説明しておきましょう:

```
% gcc gensound.c -lm     ← sin 関数を使う場合は「-lm」指定必要
% a.out | lame -rx - - | madplay -
```

`lame` や `madplay` でなくても、同様のソフトがあれば同じようにして音を聴いてみることができるはずです。

3 グラフィクス

3.1 画像の表現

音に引き続いて、今度は**画像** (イメージ) を取り上げましょう。画像を入力する装置の代表は**デジタルカメラ**や**スキャナ**、画像を出力する装置の代表は**ディスプレイ** (液晶ディスプレイや CRT ディスプレイ) と**プリンタ**ということになるでしょう。ではデジタルカメラやスキャナはどのような形で画像を取り込み、ディスプレイやプリンタはそれをどのようにして復元しているのでしょうか。

まず**モノクロ**画像から考えてみましょう。音が時間的に連続したものであったのに対し、画像は平面的な広がりを持つ、つまり空間的に連続したものです。これをデジタル化して取り込むためには、まず平面を縦横のます目に十分細かく区切ります。続いて、それぞれのます目の明るさを電圧や電流に変換する素子を使って取り込み、AD 変換してディジタル値とします。つまり、画像は縦横に並んだ多数の点の集まりであり、それぞれの点ごとに、ます目の範囲内の元画像の明るさをサンプルした値を持つわけです。なお、この「点」のことを**ピクセル** (pixel) と呼びます。ディスプレイはこの点の集まりを画面に表示し、それぞれのピクセルごとにその値に応じた明るさ/暗さで光らせます。ま

[3] ただし平均率の場合。ピアノなどは平均率でチューニングします。

[4] ちなみに $D = e^{\frac{\ln 2}{12}}$ となります。

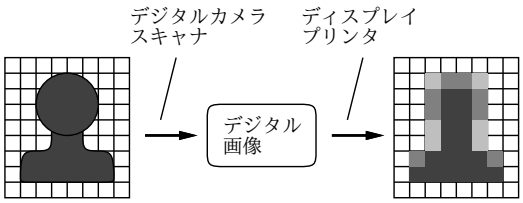

図 4: デジタル画像の原理

た、プリンタは紙の上にそれぞれの点の集まりを色素を使って定着させますが、ピクセルごとにその値に応じた量の色素を定着させます。そこで、人間がこれらを見ると元の画像 (をデジタル化して復元したもの) が見られるわけです。

カラー画像の場合も画像がピクセルの集まりであることは同じですが、色の情報を取り込むために、各ピクセルごとに赤 (Red)、緑 (Green)、青 (Blue) の光の 3 原色のフィルタを通して 3 つのサンプル値を取り込みます。

つまり、各ピクセルは 3 つの値の組 (**RGB 値**) として表現されます。広く使われているのは、RGB 値として各色ごとに 8 ビット (つまり 0〜255 の値) を使い、1 ピクセルあたり 24 ビット (3 バイト) でカラー画像を表す方法です。これを **24 ビットカラー**といいます。[5]カラーディスプレイは各ピクセルごとに RGB の 3 つの光る点を制御し、それぞれを RGB 値に応じた明るさで光らせますし、プリンタは各ピクセルごとに 3 色[6]の色素を配合して各ピクセルが RGB 値に対応する色になるよう制御します。

3.2 ピクセルグラフィクス

画像は多数のピクセルの集まりですから、画像を加工することは、個々のピクセルごとにその RGB 値を変化させてやることで行えます。また、マウスやタブレットペンなどを使って「お絵描き」をする場合も、描画範囲上でマウスポインタがなぞった部分のピクセルの色を変化させることで「インク」のようにそこの部分の色を変えることができます。

このように、画像を構成するピクセルを直接取り扱うようなグラフィクスの方式を一般に**ピクセルグラフィクス**と呼びます。ピクセルグラフィクスに基づく作画ソフトのことを**ペイントソフト** (比較的簡単な絵を描く場合に使う) や**フォトレタッチソフト** (写真などの細密な絵を加工するような場合に使う) と呼びます。Unix 上の代表的なペイントソフトとしては **xpaint**(図 5)、フォトレタッチソフトとしては **gimp**(コマンドは「`gimp-2.3`」などがあります。また、おもに画像表示に使われるコマンド **xv** にも簡単な画像加工の機能が備わっています。**ImageMagick** と呼ばれる画像処理コマンド群も画像加工に使えるコマンドを含んでいます。

ピクセルグラフィクスに基づく処理には、次のような利点があります:

- ○ たとえば画面の表示能力目一杯の細かさと色数のデータを作れば、その画面で表せるどんな画面でも表現できる。
- ○ 「ぼかし」や「にじみ」などの効果を使って中間的な色合いを持った絵や独特のタッチを持った絵が作れる。

その一方で、次のような弱点もあります:

[5]先に CSS のところで、色の指定に「`rgb(赤, 緑, 青)`」という指定を使ったことをご記憶かと思いますが、これも RGB 値を指定していたわけです。

[6]正確には、印刷の場合は「真っ黒」を表現するため 4 色目として黒の色素を持たせます。また 3 色も印刷の性質上、シアン、マゼンタ、イエロー (色の三原色) の組合せを使います。

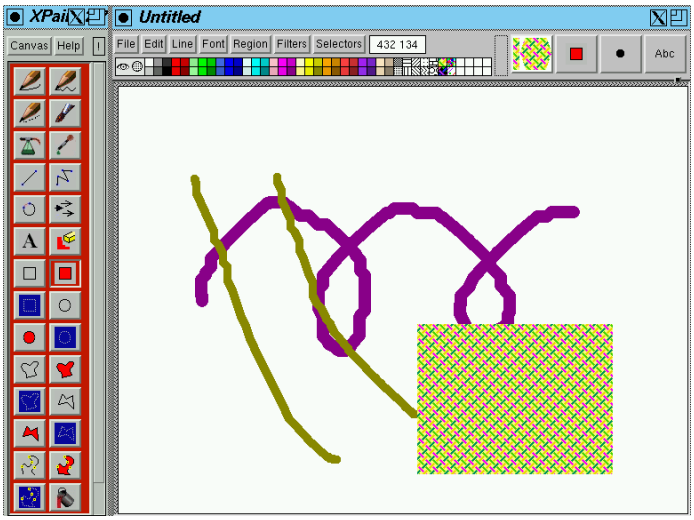

図 5: xpaint の画面

△ ピクセル数を大きくすると (これは絵を大きくする場合だけでなく、点の取り方を細かくする場合も含まれる)、ファイルも巨大になりやすい。

△ 描いた絵を拡大したり回転するなどの加工に弱く、大きくするとぎざぎざが目立ったりする。

△ ある場所に絵を「描いてしまう」と、そこのピクセルの色を設定してしまうので、後から消したり動かしたりが難しい。(消しゴムで消すというのは結局「背景で塗っている」のと同じ。そして動かすと後が「空白」になる。) [7]

いちばん最後の弱点に対しては、画像を複数の**レイヤ** (層) に分けて扱うことである程度対処可能です。レイヤー機能を持つソフトでは、透明なシート (レイヤ) を複数使ってそれぞれに絵を描き、それを全部重ねて眺めたものが最終的な絵になります。重ねる順番や位置などは描いた後でも変更できますし、描き損なった場合はそのレイヤだけ消してやり直せば済むわけです。

ここまではソフトの機能を中心に説明してきましたが、作成した画像は最終的にはファイルに出力します。ピクセルグラフィクスのファイル形式は多数あり、またソフトごとにそのソフトで扱いやすい独自形式を持ったりしますが、共通に使われる代表的なものを挙げておきます。

- **BMP**(Windows Bitmap) — Windows 固有の、圧縮のないピクセル画像ファイル。あまり使われることはない。
- **PBM**(Portable Bitmap) — Unix 文化で普及している、圧縮のないピクセル画像ファイル。形式が簡単なので後の実習で使う。
- **GIF** WWW で最初に使われた圧縮のある画像ファイル形式。色数が最大 256 色という制約があるが、アイコン等には使いやすい。
- **JPEG** — WWW で GIF に続いて普及した、写真などの画像に適した、損失のある圧縮を主に用いる画像形式。[8]
- **PNG** — WWW で最も新しく普及した形式。GIF の 256 色という制約をなくし、JPEG と異り損失のない圧縮を用いている。

Web で画像ファイルを使う場合は、最後の 3 つ (GIF、JPEG、PNG) のどれかの形式を使う必要があります (でないと、ブラウザに表示できない)。

[7] ただし、間違って塗った場合には、元の状態を記憶しておくことである程度戻すことは可能。大量には難しい。

[8] 損失のある圧縮とは、圧縮したものを展開したときに完全に元のデータとは一致しないような圧縮方式をいう。その代わり、圧縮率を高くしやすい。JPEG では圧縮率を指定することで「ファイルサイズが小さいが品質が落ちる」「ファイルサイズが大きいが品質が高い」などの制御ができる。

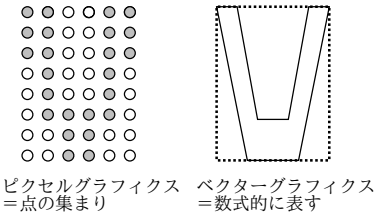

図 6: ベクターグラフィクスとピクセルグラフィクス

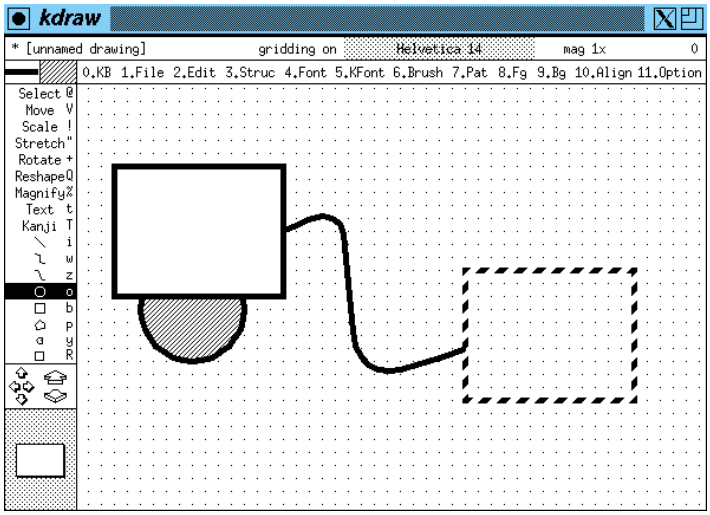

図 7: kdraw の画面

3.3 ベクターグラフィクス

ピクセルグラフィクスとは全く違う絵の表し方として、図形などの位置や輪郭を数値的/数式的に覚えておき、絵が必要になる瞬間にその式に応じて絵を生成して表示するという方式があります。このようなモデルを (位置、方向などの「ベクトル」を用いて絵を表すことから) **ベクターグラフィクス**と呼びます (図 6)。ベクターグラフィクスでは、絵は円、直線、矩形などの比較的単純な図形の集まりで表すのが普通ですが、高度なソフトになると 3 次曲線、ベジエ曲線などの数式に基づく曲線を活用してもっと柔軟な形を取り扱うこともできます。

ベクターグラフィクスに基づく作画ソフトは**ドローソフト**と呼ばれることが多いようです。Unix で多く使われているドローソフトとしては、**tgif** や **kdraw**(図 7) などがあります。ドローソフトで絵を描くのは、「無限に伸び縮み可能な針金で作った図形にスクリーントーンを貼って好きな順に重ねて行く」ようなものだと思えばよいでしょう。針金ですから、あとで自由に置き場所や大きさを調整することができるわけです。

ベクターグラフィクスの得失はだいたいピクセルグラフィクスの裏返しと考えればよいでしょう:

- ○ 図形の拡大・縮小・回転・重なり順の変更などは単にその変更に基づいて絵を表示し直すだけなのでいくらでも自由に行なえる。
- ○ 絵は数式的に表されているので、拡大してもぎざぎざになることはない。
- ○ 絵の情報は座標や形などの情報なので、ファイルの大きさは小さくて済むし、拡大/縮小してもファイルサイズは変わらない。
- △ 絵の細かさはソフトに用意されている階調機能や模様機能などで決まってしまい、細かい色合いは使いにくい。

△ ぼかし、にじみなどの効果は使えない。[9]

ベクターグラフィクスのファイル形式としては、**PostScript** があります (TeX の出力に使いましたね)。PostScript はもともとはプリンタで出力するページを記述するための言語であり、手で書くこともできます。現在広く使われている **PDF**(Portable Document Format) は PostScript の技術を土台に、よりコンパクトになるように設計されていて、印刷形式の文書を配布するのに有用です。

また、WWW などのページ内容としてベクターグラフィクスを記述できるように設計された言語として **SVG**(Scalable Vector Graphics) があります。SVG については後で簡単に紹介しますが、MSIE が SVG に対応していないため普及度はいまいちです。このため、現在でも Web 上の画像はピクセルグラフィクスによるものが大部分です。

3.4 RGB 画像を生成する

そろそろお話ばかりだとつまらないでしょうから、今度はプログラムで画像を生成してみましょう。以下に示すのは「300 × 200 の緑の背景に赤い斜め線が入っている」画像を生成するプログラムです。生成した画像はファイルに書き出す必要がありますが、ここではなるべくプログラムを簡単にするため、**PPM 形式**の画像ファイルを出力しています:

```
/* genimage.c --- create color PPM image */
#include <stdio.h>
#define WIDTH 300
#define HEIGHT 200
struct { unsigned char r,g,b; } img[HEIGHT][WIDTH];

main() {
  makeimage();
  printf("P6 %d %d 255\n", WIDTH, HEIGHT);
  fwrite(img, sizeof(img), 1, stdout);
}

pset(int x, int y, int r, int g, int b) {
  if(x >= 0 && x < WIDTH && y >= 0 && y < HEIGHT) {
    img[y][x].r = r; img[y][x].g = g; img[y][x].b = b;
  }
}

makeimage() {
  int x, y, i;
  for(x = 0; x < WIDTH; ++x) {
    for(y = 0; y < HEIGHT; ++y) {
      pset(x, y, 150, 200, 100);
    }
  }
  for(i = 0; i < 100; ++i) {
    pset(i, i, 255, 0, 0);
  }
}
```

[9]図形を塗りつぶすときに、階調 (グラデーション) や模様 (テクスチャ) などを使うことはできます。

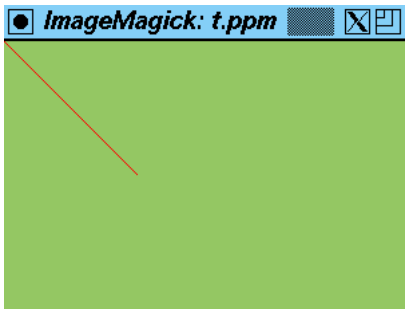

図 8: ビットマップ出力プログラムの結果

定数 WIDTH と HEIGHT は画像の幅と高さを表しています。struct... の行は配列 img の宣言ですが、buf の 1 要素は r、g、b 各 1 バイトのフィールドから成るレコード型で、それが 200 × 300 並んだ 2 次元配列が img なわけです。

main() では、まず makeimage() を読んで適当な画像を作ります。この中は後で自由に直してみてください。それが終わったら、PPM 形式ファイル用の「ヘッダ行」(画像形式と幅/ 高さの情報) を出力し、続いて配列全体を書き出します。PPM 形式の場合、ここにあるように、続いて img の内容をそのまま全部書くだけでよいのです。

次の関数 pset() は、単に渡された x と y の値が画像の範囲内に入っているかどうかチェックして、OK の時はその位置の img の要素に渡された RGB 値を設定するだけです。このような下請け関数を用意しておくだけで、プログラム本体がずっと見やすくなります。

最後の makeimage() が画像を用意する中心部分です。ここではまず、x を 0〜WIDTH、y を 0〜HEIGHT の範囲で変化させながら、つまり画像全体を $(250, 200, 100)$ の色に塗りつぶしています。次に i を変化させるループで、$(0, 0), (1, 1), \ldots (99, 99)$ の点 (つまり斜めの線上の点) について、色を真っ赤に設定しています。

このプログラムが genimage.c に入っているとして、それを動かすには次のようにしてください:

```
gcc genimage.c
a.out >test.ppm
display test.ppm
```

このようすを図 8 に示しました。なお、表示用プログラムは display 以外に xv など、PPM 形式を表示できるものなら何でも構いません。

このようなプログラムを動かしてみると、画像データというのは単に「RGB 値を山のように並べただけのもの」だというのが納得できると思うのですが、いかがでしょうか。

3.5 PostScript — ベクターグラフィクス記述言語

先に説明したように、PostScript はベクターグラフィクスのファイル形式ですが、実はファイル形式というより「言語」でもあります。今回は、PostScript ファイルは全体として次のような形で作ります。

```
%!PS-Adobe-2.0                ← PostScript であることを表す
%%BoundingBox: 0 0 400 300   ←絵の範囲を表す。今回は 400x300 にした。
…(ここにさまざまな図形記述を入れる)…
showpage      ←プリンタに送った場合にページを出力する
```

前回やったことですが、線を引くには次のコマンドを使います。

- `newpath` — 新しい線引きを開始する。
- X Y `moveto` — ペンを指定した座標 (X, Y) に移動。
- X Y `lineto` — 座標 (X, Y) までの線を登録しながら移動。
- `stroke` — `lineto` で指定した線引きを一気に実行する。
- W `setlinewidth` — 線の太さを Wpt にする。

では正方形を描くという簡単な例を挙げておきます (図 9)。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 400
newpath 100 100 moveto 100 200 lineto
200 200 lineto 200 100 lineto 100 100 lineto stroke
showpage
```

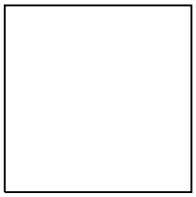

図 9: 正方形を描く

ところで、PostScript は言語だと書いたのはどういうことでしょう? それは、たとえば「ループ」を使って先の正方形を繰り返し描いたりできます。同じ場所に描いてもつまらないので、次のコマンドも知っておいてください。

- T_x T_y `translate` — 絵を描くときの原点を X 軸方向に T_x、Y 軸方向に T_y だけずらす。
- S_x S_y `scale` — 絵を描くときの大きさを X 方向に S_x 倍、Y 方向に S_y 倍する。
- D `rotate` — 絵を描くときの角度を原点のまわりに (反時計回りに)D 度回転する。

繰り返し自体は次のようにします。

- N { 動作列 } `repeat` — 「動作列」を N 回繰り返す。

では、これを使って正方形を 5 回ずらして描いてみます (図 10)。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 400
5 {
  newpath 100 100 moveto 100 200 lineto
  200 200 lineto 200 100 lineto 100 100 lineto stroke
  30 -15 translate
} repeat
showpage
```

最後に、フォント (文字) について説明しましょう。文字を表示するのには、次の 2 つのステップが必要です。

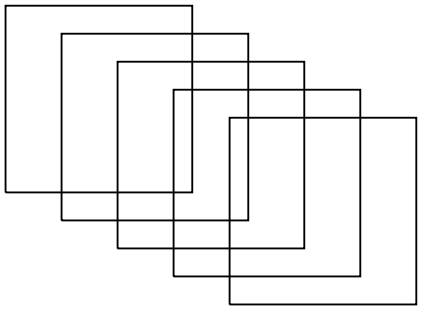

図 10: ループで正方形を描く

- /フォント名 `findfont` S `scalefont setfont` — 指定した名前のフォントを探してきて、サイズ (ポイント数)S に設定し、使用するフォントとしてセットする。
- X Y `moveto` (文字列) `show` — 指定位置 (X, Y) に文字列を表示する。

一度フォントをセットしたら、別のものに変更しないで繰り返し `show` を使って構いません。文字列を丸かっこで囲んで指定するというのはちょっと変わっていますね。もし文字列の中に丸かっこを入れたければ、「`\(`」「`\)`」のようにバックスラッシュを前につけてください。

図 11: さまざまなフォント

では例を見てもらいますが (図 11)、途中で「灰色の濃さ」や「色」を変更しているのでその説明もしておきます。

- G `setgray` — 灰色の明るさ (0.0〜1.0、大きいほど明るい) を設定。
- R G B `setrgbcolor` — 色を RGB で指定 (どれも 0.0〜1.0、大きいほどその色が強い)。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 300
/Helvetica findfont 24 scalefont setfont
20 20 moveto (This is Helvetica 24pt.) show
/Times-Roman findfont 36 scalefont setfont
20 60 moveto (Times\(-Roman\) 36pt.) show
/Courier-Bold findfont 12 scalefont setfont
20 110 moveto (This is Courier-Bold 12pt.) show
```

```
0.5 setgray
/Ryumin-Light-EUC-H findfont 40 scalefont setfont
20 140 moveto（これは明朝体です。）show
5 rotate 0.8 0.3 0.2 setrgbcolor
/GothicBBB-Medium-EUC-H findfont 48 scalefont setfont
20 180 moveto（ゴシック体です。）show
showpage
```

なお、日本語については必ずEUCコードを使用してください。[10]Emacsであれば「`Ctrl-X RET f euc-jp RET`」でファイルの文字コードをEUCに設定できます。

ところで、PostScriptのフォントは**アウトラインフォント**、つまり数式で輪郭を表現したフォントを使っていることが特徴で、このため任意のポイント数に大きさを設定できます。今日では他のソフトでもこれが普通ですが、PostScriptがこれを始める前は、フォントはすべて「ある大きさで」デザインされていたので、自由にさまざまなサイズを指定することはできませんでした。実際にさまざまな大きさにフォントを変更してみてください。

3.6 TeX文書に画像を入れる OPTION

せっかく画像が作れるようになったので、これをTeX文書に入れる方法を説明しておくことにします。

1. TeXに取り込むためには、ファイル形式はEncapsulated PostScript形式 (`%%BoundingBox:`の指定されたPostScript形式) でなければなりません。先に作ったPostScriptの例題は既にそのようにしてありましたね。それ以外の形式のファイルであれば、次のようにして変換できます。

   ```
   convert test.ppm fig1.ps
   ```

   `convert`コマンドはファイル名の拡張子部分を見て適切な形式への変換を行なってくれます。[11]

2. `\documentclass`と`\begin{document}`の間に次のような行を追加します:

   ```
   \usepackage{graphicx}
   ```

3. 本文中の図を入れたい場所に次のようなコマンドを入れます:

   ```
   \begin{center}
   \includegraphics[scale=0.5]{fig1.ps}
   \end{center}
   ```

   見て分かる通り、「0.5」は縮小比率、「`fig1.ps`」はPSファイル名です。scaleの倍率の代わりに`width=8cm`のようにしてでできあがりの幅を指定することもできます。

あとはこれまで通りに`platex`を使ってください。`platex`を使うときだけでなく、`xdvi`や`dvips`を使うときも、そのPSファイルが同じ場所に置いてある必要があります。

[10]日本語フォントとしてEUCフォントを指定しているためです。
[11]また、PostScriptがEPS形式でない場合は`ps2eps`コマンドを使うことでEPS形式に変換することができます。

4　3D とアニメーション

4.1　3 次元グラフィクス OPTION

ここで少し寄り道して、立体的な描画、つまり **3 次元グラフィクス** (3D) についても簡単に説明しておきましょう。3D の描画の何が 2D(2 次元グラフィクス、平面的な画像) と違うのでしょうか? 3D の絵を「絵描きさんが立体的な絵を描くように」ペイントソフトで描くことも不可能ではありませんが、手間が掛かってあまりよい方法とは言えません。[12]

そうする代わりに、計算機の内部に 3 次元的な物体の形状データを作成し、その形状データを特定視点から見た時のようすを計算によって作り出すことで 3D の描画を行うのが普通です。この方法であれば、色あいや視点などを調整したいときには後半の計算だけやり直せば済むという利点があります。

ちなみに、前半の 3 次元の形状データを作り出すことを**モデリング**、

後半の 3D モデルから画像を生成することを**レンダリング**と言います。前半はたとえばシミュレーションなどの計算で立体空間の様子を作り出すこともできますが、典型的には**モデリングツール**と呼ばれる一種のエディタで人間がさまざまな物体の 3D モデルを作成します。

一方、後半のレンダリングには、主要なやり方としては次の 2 つがあります:

- **ポリゴンレンダリング** — 3 次元の物体の曲面を図 12 のように多面体で一担近似し、その各面ごとに「物体の色」「光源 (太陽やランプなど) と面の角度に応じた反射」などを計算し、その面を画面上に投射した範囲を計算した色で塗る。ただしその後、

  各面のつながりをスムーズにするよう平均などを取って調整する。

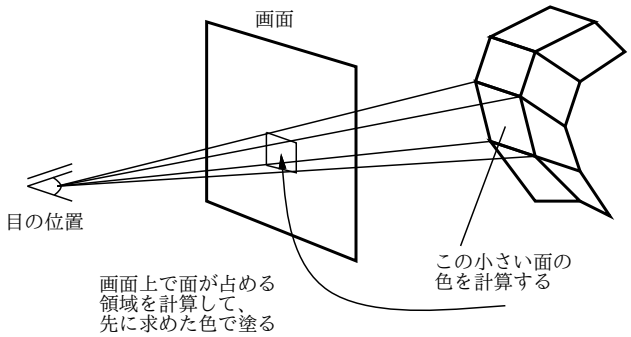

図 12: ポリゴンレンダリングの原理

- **視線追跡** (**レイトレーシング**、lay tracing) — 目の位置から画面上の各ピクセルを結んだ線を延長し、3 次元モデルの物体にぶつかる位置を求める。その位置での物体の色合いを求め、さらに物体が「つるつる」なら線を「反射」させてその先へ進む。物体がレンズみたいに「(半) 透明」なら反射光と屈折光の両方を処理する。その結果求めた「色」でそのピクセルを塗る。これを画面上の全ピクセルについて行う (図 13)。

これらを比べると、視線追跡の方が (反射や屈折まで扱うから) ぐっとリアルな絵が作れますが、計算量は膨大になります。一方、ポリゴンレンダリングは多面体近似の荒さによって計算の節約がコントロールできます。最近のグラフィクスワークステーションやゲームマシンでは、この計算を行うための専用ハードウェアを備えていて、これによって高速に 3 次元画像を生成しています。

[12] もちろん、デジカメで立体的な写真を撮影するのは簡単ですが、これは 3D グラフィクスとは普通呼びません。

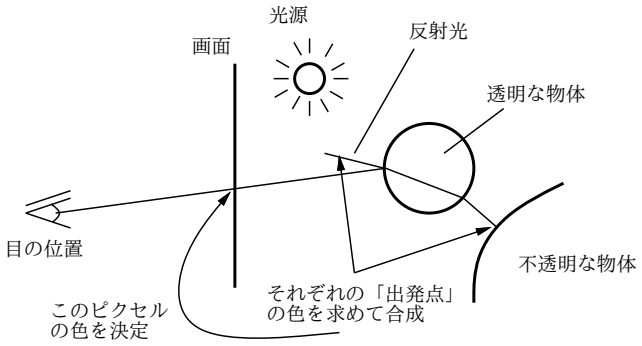

図 13: レイトレーシングの原理

4.2 動画とアニメーション OPTION

ここまではすべて、動かない 1 枚の画像について考えて来ましたが、**動画** (**ムービー**) についても簡単に触れておきましょう。実は動画の原理は非常に簡単で、(既に多くの人がご存じだと思いますが)1 秒間につき 20 個とか 30 個の画像を取り込み、それらを短い時間間隔で順次表示すれば、絵が動いて見えます (この動画を構成する 1 つひとつの画像のことを**フレーム**と呼びます)。

ここでおもな問題は、1 つの画像ですらそれなりのデータ量になるのに、それを毎秒 20 も取り込むとデータ量が膨大になるという点にあります。このため、1 つずつのフレームを圧縮して保管したり、連続したいくつかのフレームについて、前のフレームとの**差分** (違っている部分) だけを残すなどの方法で、データ量を圧縮します。その処理を行うのにも 1/20 秒間隔で処理しないと間に合わない (次のフレームが来てしまう) ので、ハードウェア的に処理を行うことが多くなっています。音と同様、動画についても代表的なファイル形式は **MPEG**、**QuickTime**、**WindowsVideo**(WMV) など複数あります。

ここまでは画像を取り込んで動画ファイルにするという話でしたが、画像を生成して動画をつくり出す、つまり計算機による**アニメーション**生成も多く行われています (この手法で商業映画も作られていますね)。ごく少ない枚数のフレームなら、手で絵を描いてそれをパラパラ漫画のように表示させることでアニメーションさせることもできます。しかしもちろん、1 枚ずつコマンドを指定したり人間が手で描くのでは長いものを作るのはとても大変です。そこで、モデルに基づくアニメーションが多く使われます。これは上で説明したモデルに基づく描画の応用編で、計算機の内部にモデルを用意して、そのモデルを時間とともに変形させていき、それぞれの時点のモデルから画像を生成することで動画が生成できわけです。この場合、モデルが 3 次元なら 3 次元のアニメーション、2 次元なら 2 次元のアニメーションということになります。

アニメーションはさらに、実時間的/対話的なものとそうでないものがあります。たとえば、3D モデルから高品質な画像を作り出すのはかなりな計算量が必要なので、3 分間の動画を計算するのに数時間の計算を費す、ということもあり得ます。この場合、完成した動画は美しいかも知れませんが、映画のように「鑑賞」するだけで再生時に変化するような点はないわけです。

一方、計算が十分高速である場合 (または計算が追い付く程度まで画像の品質を妥協した場合)、3 分間の動画の計算が 3 分間より短くて済むことになります。そうなると、画像を「計算しながらその場で上映」でき (**実時間アニメーション**)、ユーザが何か操作をするとその操作に応じて動画を変化させる (**対話的グラフィクス**) ことができます。

もちろんこれは**テレビゲーム機**で行なわれていることであって、今や珍しいことでもなんともありませんが、その「計算を追い付かせる」ためにさまざまなハードウェア上/ソフトウェア上の工夫がなされています。[13]

[13] PlayStation(初代) が出るまでは対話的 3 次元グラフィクスは極めて限られた人にしか体験できませんでした。コンシュマー技術恐るべし。

5 Web 上の描画: SVG と canvas

5.1 Web の発展と HTML5

ここまで、Web ページの記述は今日の標準である HTML4.01 に基づいて説明してきましたが、この「先」についても簡単にお話しておきます。まず、HTML などの標準を取りまとめている W3C(WWW Consortium) では HTML ではタグの種類が固定であるためさまざまな用途に対応できないとうい問題を克服するため、タグを自由に定義できる汎用的なマークアップ言語 XML(eXtensible Markup Language) を制定し、これが広く使われるようになりました。XML が HTML と違っている主要なこととして、次のものが挙げられます。

- タグ名は必ず小文字で書くこととし、属性値は必ず「"…"」で囲むこととした。
- 中身のある要素のタグは「<x>…</x>」のように必ず対で書き、閉じタグの省略は許さないこととした。逆に中身のないタグは「<x />」のようにそれ単独であることを示す形とした。
- 名前空間機能により、複数の XML に基づく言語を混在させられるようにした。

XML は現在では、情報システムのさまざまな分野でさまざまな情報を記述する言語として普及しています。

W3C では XML の普及を受けて HTML も XML に移行することを計画し、まず HTML 4.01 をおおよそそのまま XML の形に合わせた XHTML 1.0 を制定しました。XHTML 1.0 は HTML 4.01 とさほど違わないのでそれなりに使われています。

しかしその後、W3C は引き続きより XML 色の強くなった XHTML 1.1、XHTML 2.0 を推進しようとしましたが、各ブラウザベンダーなどはこれまでと互換性がなく、機能としてはあまり魅力的でなかったこれらの標準には否定的で、実装する主要ブラウザが現れないまま年月が経過しました。

その中で、2004 年に Opera、Mozilla、Apple が WHATWG(Web Hypertext Applicaiton Technology Working Group) を構成し、後方互換性がありながらマルチメディアなど魅力的な機能も含まれる HTML の後継仕様を検討するようになりました。

このような状況を見て、W3C も 2007 年にこの WHATWG の方向性を共有することにして、その考え方に基づき HTML5 を策定することとなりました。HTML5 では HTML 4.01 に足りなかった意味的なマークアップを強化するとともに、音声、ビデオ、図形などをこれまでより簡単に取り込める機能も追加した仕様を策定しています。

現状では HTML5 はまだ検討段階で、それに完全対応したブラウザももちろんありませんが、ここではせっかく図形の話をしたので、XML の構文を用いてベクターグラフィクスを表現する SVG と、スクリプトによるグラフィクスの描画や操作を可能とする機能である canvas について紹介します。

これらはいずれも、HTML5 では<svg>...</svg>、<canvas>...</canvas>というタグで表すことになりますが、我々が実習に使っている Mozilla ブラウザでも (多少工夫が必要だったり機能が制限されていますが) 動作させてみることができます。

5.2 SVG

SVG(Scalable Vector Graphics) とは上で説明したように、XML の構文を用いてベクターグラフィクスを表現する記述言語で、HTML や XML と同じく、W3C が標準を定めています。その中身の話に入る前に、Mozilla ブラウザで SVG を表示させる方法を説明しておきます。Mozilla には SVG 表示機能が組み込まれているのですが、通常の HTML からだと<svg>…</svg>タグを認識しないため、別のファイルに分けて「これは SVG だよ」と別途教える必要があります。このために、コンテンツ種別を指定して別ファイルを埋め込む object 要素を使用します。

```
<!DOCTYPE html>
<html>
<head>
<title>SVG example</title>
</head>
<body>
<object width="500" height="400" data="test1.svg" type="image/svg+xml">
(sorry, cannot display SVG on your browther...)
</object>
</body>
</html>
```

1 行目の DOCTYPE 宣言がこれまでと変わっていますが、これだけの指定だと「最新の HTML」という意味になるので HTML5 を表します (もっともこのファイルでは HTML5 固有の機能は何も使っていないので HTML 4.01 を指定しても構いません)。そして、ページ本体には object 要素だけが含まれています。

- `<object width="幅" height="高さ" data="`URL`" type="データ種別">`内容…`</object>` — 幅と高さを指定し、その大きさの領域内に URL で指定した内容を埋め込む。内容の種別 (MIME タイプ) を「データ種別」で指定。

ここでは別に作成する SVG ファイル名を `data` 属性に指定し、`type` 属性にその種別として「`img/svg+xml`」(XML 表現された SVG) を指定しています。なお、「内容…」の部分は SVG が表示できない環境の場合に代わりに表示されるので「このブラウザでは SVG が表示できません」と書いてあるわけです。本来なら、SVG の代わりに同じ内容の図を画像で見せるため img 要素を入れておく、などの方がより親切でしょう (このような、うまく行かない場合の代替コンテンツのことを「フォールバックコンテンツ」と呼びます)。

ではいよいよ、`test1.svg` の中身である SVG を見てみます。

```
<svg width="500" height="400" xmlns="http://www.w3.org/2000/svg">
  …中身…
</svg>
```

SVG なのでトップレベルは`<svg>...</svg>`となっています。単独の XML ファイルでは中身がどのような XML なのかを通知する必要があるので、そのために「`xmlns="`URL`"`」という属性を使用します。ここでは SVG を表す URL を指定しています。あと、幅と高さは先に object 要素で指定したのと一致させています。

SVG による図形記述の中身は、基本となる図形を組み合わせて行くことです。基本図形として次のものがあります。

- `<rect x="`X`" y="`Y`" width="`W`" height="`H`" />` — 左上隅の XY 座標と幅と高さを指定して、長方形を描く (「`rx="横幅" ry="縦幅"`」を指定することでその幅だけ角を丸めた長方形にできる)
- `<circle cx="`X`" cy="`Y`" r="`R`" />` — 中心の XY 座標と半径を指定して、円を描く
- `<ellipse cx="`X`" cy="`Y`" rx="`R_x`" ry="`R_y`/>` — 中心の XY 座標と横半径、縦半径を指定して、楕円を描く
- `<line x1="`X_1 `y1="`Y_1`" x2="`X_2`" y2="`Y_2`" />` — 2 点の XY 座標を指定して、線分を描く
- `polygon points="`$X_1\ Y_1\ \cdots\ X_n\ Y_n$`" />` — N 個の頂点の XY 座標を指定して閉じた多角形を描く

- polyline points="X_1 Y_1 $\cdots$ X_n Y_n" /> — N 個の頂点の XY 座標を指定してそれらを通る折れ線を描く
- path d="指定…" — PostScript の path と同様の方法でパスを描く。「M x y」→「newpath x y moveto」と同じ、「L x y」→「x y lineto」と同じ、「Z」→「closepath」と同じ、小文字の m と l は rmoveto、rlineto に対応
- <text x="X" y="Y">文字列</text> — 文字列を表示
- <g>...</g> — 中に指定した複数の図形をグループにする

これらは図形の「形」を指定するだけですが、色などの見え方はこれえらのタグに style 属性をつける (か、または HTML と同様に CSS を別途書く) ことで指定します (g 要素を使うことで、複数の図形にまとめてスタイルが指定できることにも注意)。その中身は次のものがあります。

- stroke: 色 — 線の色を指定
- stroke-width: 長さ — 線の幅を指定
- stroke-opacity: 値 — 0.0〜1.0 の値で透明〜不透明を指定
- fill: 色 — 内部の色を指定。none だと塗らない
- fill-opacity: 値 — 0.0〜1.0 の値で透明〜不透明を指定
- font-family、font-size、font-weight、font-style、text-decoratin — text 用で、意味は HTML+CSS と同じ

また、これまた PostScript と同様、移動/拡大縮小/回転が指定できます。その場合はタグに transform 属性をつけて、次の指定を 1 つ以上入れます。

- translate(X,Y) — (X, Y) だけ並行移動
- scale(S)、S 倍に拡大。倍率を 2 つ指定することで X 方向と Y 方向を別の倍率にもできる
- rotate(D)、rotate(D,X,Y) — 角度 D だけ回転

回転では、XY 座標を指定することで「その座標を中心に回転」が指定できます (これは PostScript で不便だったところなので、けっこう便利)。

では、これらを使った例を見ていただきましょう (図 14)。これをたとえば test1.svg というファイル名でサーバに置き、先の HTML から参照するわけです。

```
<svg width="500" height="400" xmlns="http://www.w3.org/2000/svg">
 <rect x="10" y="10" width="60" height="40" style="fill: pink" />
 <rect x="80" y="10" width="60" height="40" rx="10" ry="10"
    style="fill: none; stroke: blue; stroke-width: 4px" />
 <g style="fill: none; stroke: green; stroke-width: 4px">
  <circle cx="180" cy="50" r="25" />
  <ellipse cx="260" cy="50" rx="40" ry="20" />
  <polygon points="350 10 310 90 390 90" />
  <path d="M 390 20 l 20 60 l 20 -60 20 60 l 20 -60 20 60 Z" />
 </g>
 <g style="font-size: 40px">
  <text x="10" y="150">This</text>
  <text x="120" y="150" transform="rotate(60,120,150)">is</text>
  <text x="100" y="90" transform="scale(2.0)">a pen</text>
 </g>
</svg>
```

- 最初の長方形はピンクに塗る。
- 2 番目の長方形は青線で描き、角を 10 ピクセル丸める。
- 以下 4 つの図形はまとめて緑線で描く。
- 円、楕円、三角形、ジグザグを閉じた図形を描く。
- 以下 3 つのテキストは 40 ポイントのフォント使用。
- 「This」はそのまま。「is」は起点中心に 60 度回転。「a pen」は拡大。

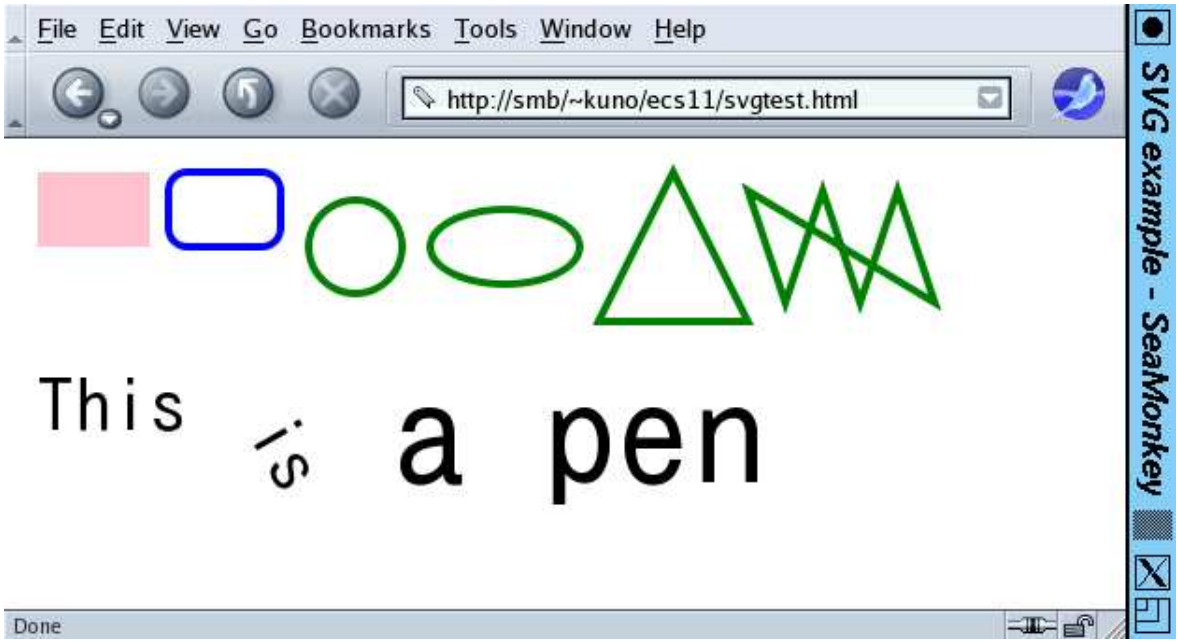

図 14: SVG による表示の例

なお、SVG のデータ構造は HTML と同様に DOM を通じて JavaScript から操作できます。簡単な例として、1 つの四角をクリックすると、他の四角の大きさが変化する、という例を示しておきます。

```
<svg width="500" height="200" xmlns="http://www.w3.org/2000/svg">
<script type="text/javascript"><![CDATA[
function change() {
  var r1 = document.getElementById('r1');
  r1.setAttribute("width", 80); r1.setAttribute("height", 100);
}
]]></script>
 <rect x="10" y="10" width="60" height="40" style="fill: pink"
   onclick="change()" />
 <rect id="r1" x="80" y="10" width="60" height="40" rx="10" ry="10"
    style="fill: none; stroke: blue; stroke-width: 4px" />
</svg>
```

これは、最初の四角に `onclick` 属性を指定して JavaScript の関数 `change()` を呼び出し、その中で `document.getElementById()` を使って `id="r1"`と指定した 2 番目の四角を取り出して、その属性 2 つを `setAttribute()` で変更しています。

なお、JavaScript コードがヘンなもので囲んでありますが、これは XML の中にそのまま別のものを入れることはできず、CDATA セクションという形で囲む必要があるという制約のためです (あんまり気にせず、XML とはこういうものなのだと思ってください。

5.3 canvas OPTION

SVG が W3C で標準化された XML に基づくグラフィクスの記述言語であるのに対し、canvas は WHATWG で取りまとめられ、「ブラウザ上で簡単に絵を描く」機能を提供しています。以前は JavaScript でブラウザ画面の中のさまざまな要素を操作する (消したり動かしたりする) ことはできても、「絵を描く」ことはできず、そのような機能がとても求められた結果作られた、という感じです。こちらも Web ページへの入れ方から見てみましょう (とりあえず四角を 1 個描いています。

```
<!DOCTYPE html>
<html>
<head>
<title>canvas example</title>
<script type="text/javascript">
var canvas, ctx;
function init() {
  canvas = document.getElementById('c0');
  ctx = canvas.getContext('2d');
  ctx.fillStyle = 'rgb(200,220,255)';
  ctx.fillRect(20, 20, 80, 60);
}
</script>
</head>
<body onload="init()">
<canvas id="c0" width="400" height="300">
(canvas not supported on your browser...)
</canvas>
</body>
</html>
```

Mozilla は canvas タグを認識するので、こちらはファイルを分ける面倒はありません。そもそも canvas は XML ではないし中身も空っぽなので別ファイルに分けて読み込ませるような内容はありません。では中身はどうするかというと、すべて JavaScript で描画します。ここではページがロードされ終ったら `init()` という関数を呼び出し、その中ですべての処理を行っています。その最初に必要なのは (1) HTML 中の canvas オブジェクトを `document.getElementById()` で取得し、(2) canvas に描画するためのコンテキストオブジェクトを `getContext('2d')` で取得することです (現在は種別としては 2 次元グラフィクスを表す「2d」しか使えません)。

この後はすべて、このコンテキストオブジェクトに対するメソッド呼び出しやプロパティ設定でスタイルの設定や描画を行います。その中身はここまでに PostScript や SVG で見て来たのとよく似ています。図形を描くメソッドは次のものがあります。

- `fillRect(`X`,` Y`,` W`,` H`)`、`strokeRect(`X`,` Y`,` W`,` H`)`、`clearRect(`X`,` Y`,` W`,` H`)` — 長方形の領域を塗る (ないし、輪郭を描く、ないし領域をクリアする)
- `fillText(`文字列`,` X`,` Y`)` — 文字列を描画

- beginPath()、moveTo(X, Y)、lineTo(X, Y)、arc(X, Y, R, D_1, D_2, B)、closePath()、stroke()、fill() — PostScript の同名のものと同様。arc については円弧の中心の XY 座標、半径、開始点と終了点の角度 (ラジアン)、true か false かで描く向き (右廻りか左廻り) を指定します
- drawImage(IMG, X, Y)、drawImage(IMG, X, Y, W, H) — 画像を描く (後者では幅と高さを指定することで引き延ばしや縮小も可能)

画像は通常のブラウザ上の画像オブジェクトを指定します (たとえば img 要素を取って来るなど)。画像が描けることからも分かるように、canvas はベクターグラフィクスではなく、単に「任意の絵が描けるような API(呼び出しインタフェース)」なわけです。

図形等を描く再のスタイルの設定はプロパティの代入で行います。一方、座標変換はメソッドで指定し、PostScript 等と同様です。

- fillStyle、strokeStyle、lineWidth、globalAlpha — 塗りつぶしの色、線の色、線の幅、描画全体の透明度を設定[14]
- font — 文字のサイズとフォントを「'20px Times New Roman'」などのように指定
- translate(X, Y)、rotate(R)、scale(S_x, S_y) — 移動、回転、拡大縮小
- save()、restore() — 描画状態全体をスタックに保存し、またスタックから復元

このほか、ゲームやユーザによるお絵描きを行うためにはマウスイベント等を扱う必要がありますが、こちらは通常のブラウザのイベント機構を使います。上の例をさらに直して、画面上をクリックするとその場所に円が現れるようにしてみました (図 15)。

```
<!DOCTYPE html>
<html>
<head>
<title>canvas example</title>
<script type="text/javascript">
var canvas, ctx;
function init() {
  canvas = document.getElementById('c0');
  ctx = canvas.getContext('2d');
  ctx.fillStyle = 'rgb(200,220,255)';
  ctx.fillRect(20, 20, 80, 60);
  addEventListener('mousedown', mousedown, false);
}
function mousedown(evt) {
  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.beginPath();
  ctx.arc(evt.layerX, evt.layerY, 10, 0, Math.PI*2, true);
  ctx.fill();
}
</script>
</head>
<body onload="init()">
<canvas id="c0" width="400" height="300" style="position:relative">
```

[14] そのほか個々の色も、rgba(R,G,B,A) という書き方で A のところに 0.0〜1.0 の値を設定することで透明度のある色が指定可能です。

```
(canvas not supported on your browser...)
</canvas>
</body>
</html>
```

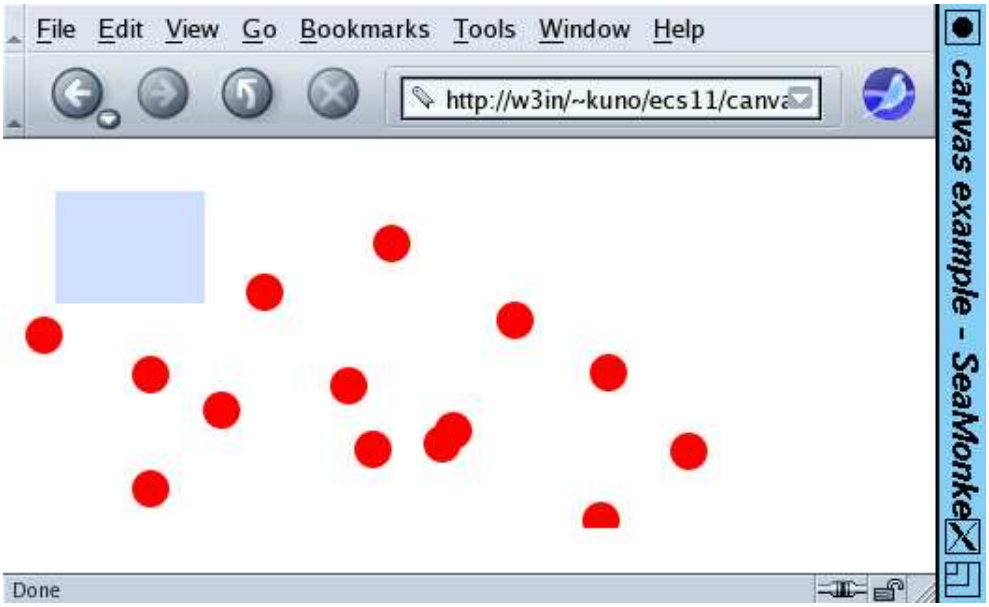

図 15: canvas による描画の例

6 まとめと演習

この章の前半では、アナログとデジタルの違いについて改めて考え、続いて私達の身のまわりにある代表的なアナログ情報である音や画像のデジタル化や計算機内部での表現方法についてその原理を中心に見てきました。絵や音が計算機上でどのようにして扱われているのか感じがつかめたかと思います。引続き後半では、前回学んだ HTML と CSS の機能を土台として、実際のサイトを作るプロセスと、ページ設計のやりかた、レイアウトのつけかたについて取り上げました。

5-1. `gensound.c` をコピーしてきて、まずそのまま動かしてみなさい。続いて、何らかの「メロディ」か「和音」を自力で作ってみなさい。「雑音」はメロディでも和音でもないことに注意。

5-2. `gensound.c` のプログラムを手直しして、「連続的に周波数が変化する音」を作ってみなさい。連続的な変化であればその内容は問いませんが、「でたらめな音」「雑音」は連続的な変化とは言えないことに注意。

5-3. `genimage.c` をコピーしてきて、まずそのまま動かしてみなさい。続いて、図形を何らかの形で違ったものにしさい。たとえば直線でなく曲線にする、線を太くする、背景を均一色でなく少しずつ色が変化するようにする、などが考えられます。「でたらめにプログラムを変更した結果」は図形ではないことに注意。

5-4. `genimage.c` のプログラムを手直しして、「連続的に色彩が変化するような画像」を作ってみなさい。連続的な変化であればその内容は問いませんが、「でたらめに変えた結果」では連続的な変化にはならないと思われます。

5-5. PostScript の例題を打ち込んで動かしてみなさい。様子が分かったら、次のどれかをやってみなさい。

a. 簡単な図形を繰り返し表示させて模様を作る。
b. 文字と必要なら簡単な図形 (横線とか) を使って自分の名詞を作る。
c. その他自分が描いてみたいと思うもの何でも。

5-6. SVG を使って絵を描くページ、または canvas を使って絵を描くページのサンプルをコピーしてきて表示させてみなさい。表示できたら、その内容を編集して新しい図形 (既にあるのや例題にあるものとは違う形のもの) を何か追加してみなさい。

7 最後に

「計算機科学」では「計算機科学基礎」に引き続き、コンピュータの世界で扱われる (現に私たちが毎日恩恵を被っている) さまざまな技術が、どのような原理に基づいていて、それらの機能やサービスの内実はどのように実現されているのかを、毎回話題を選んで取り上げて来ました。

これらの原理的な内容は「無駄な知識」「知らなくても使えればよい」などと言われて軽視されがちです。しかし、皆様がこれらの原理や技術を知っておくことは、これらの原理や技術に基づくシステムを計画したり取り扱ったりするときに、根源となる判断を誤らないための、重要な「羅針盤」となるものと考えています。ぜひ、これからもこの方面での勉強を続けて頂きたいと思います。

では 5 回に渡り (「計算機科学基礎」から通せば 10 回に渡り)、どうもおつかれ様でした。