記 号 処 理 58－2
（1991 1．11）

Yy へのカラーグラフィック機構の組込みについて

太田幸雄†、田中啓介†、古坂孝史‡、井田昌之†

†青山学院大学、‡青山学院大学/CSK

Yy ウィンドウツールキットの実現である *YyonX* に対して、カラーイメージ処理とアニメーション処理について設計と試作を行なった。これに関する機構について論じる。

YyonX ver1.2 でのカラーグラフィック機構の問題点を整理し機能拡張をし ver1.3 を作成した。さらに、描画演算を用いたカラーイメージの合成で起こる色化けの対処方法として、システム予約として透明色を導入することにより任意形状のカラーイメージの重ね合わせと動的な移動を可能とした。また、アニメーションを実現する1手法として *Yy* サーバのテリトリにイメージバッファを持たせ、それを一定間隔で切り変える機構を試作した。

A Color Graphic Extension for *Yy*

Yukio Ohta*, Keisuke Tanaka, Takashi Kosaka, Masayuki Ida
CSRL, Information Science Research Center
Aoyama Gakuin University
4-4-25 Shibuya, Shibuya-ku, Tokyo JAPAN 150
*yohta@green-hill.csrl.aoyama.ac.jp

This paper discribes a design and an implementation of a color imaging system and an animation system with a color graphic extension for *YyonX* ,which is an implementation of *YyWindow Tool Kit* on X window system. We analyzed issues of the color graphic system for *YyonX* ver1.2 and implemented an extended color graphic system for *YyonX* ver1.3. A transparent-color is introduced to put and dynamically move a free figure image on another image. And for realizing animation, an image buffer is implemented in *territory* of *Yyserver*, and an image in the image buffer is displayed by turns.

1 はじめに

Yy ウィンドウツールキット [1] は Common Lisp 用ポータブルウィンドウツールキットである。*YyonX* [2][3] は *Yy* を X window 上に *Yy* サーバと *Yy* クライアントの 2 つのプロセスとして実現したものである [4]。*YyonX* におけるカラーイメージ処理とアニメーション処理の実現方法と試作について述べる。

YyonX ver1.2 はカラーグラフィックスの機構をそなえており、線や円弧の描画はＲＧＢの値による色の指定を行なうことが可能である。カラーイメージの描画は点毎に色を指示して行なう。

Yy ではテリトリと呼ぶ矩形領域に対して描画を行なう [5]。従ってイメージの表示や合成はイメージの形状に関わりなく全て矩形領域で行なう。複数のテリトリの一部分または全部を重ねた状態で表示すると下に隠れた部分は表示されない。また、テリトリのカラーイメージを他のテリトリに複写する場合も矩形領域で行なう。イメージを複写する時に OR や AND などのビット演算の指定が可能であるが、ビット演算後の色は不定である。表示させるイメージは必ずしも矩形ではないため、矩形領域を基本とするオペレーションではイメージ処理に対して十分な機能を提供できない。

そこで、*YyonX* ver1.3 では透明色を導入し、カラーイメージ処理の拡張として任意形状のカラーイメージの重ね合わせ処理の実現を図った。合わせて、カラーイメージの表示や取り扱いのための機構を拡張した。また、合成されたカラーイメージの利用方法の 1 つとして、複数のカラーイメージを用いたアニメーション機構の試作を行なった。

2 カラーイメージ処理の拡張について

2.1 *YyonX* ver1.2 でのカラーの扱いとその問題点

YyonX ver1.2 ではＲＧＢの値によりカラーを指定する。color は Red、Green、Blue のスロットを持つクラスであり、関数 `make-color` がそのクラスのインスタンスを生成する。利用者は color のインスタンスを用いて任意の色の線や円の描画が可能である。内部的には *Yy* クライアントがＲＧＢの値を *Yy* サーバに送り、サーバ側では指定されたカラーを使用可能にしてそのカラー番号をクライアントに返す。以降サーバとクライアント間ではこのカラー番号を用いてカラーの指定を行なう。(図 1)

単純に線や円を描画するだけであるならばこの仕様で支障はない。またカラーイメージの描画についても 1 点づつ位置と色を指定して描画可能である。しかしその性能は実用的ではない。

YyonX ではカラーイメージを切り貼りして表示する方法が 2 つある。1 つは、あるテリトリのイメージデータの一部分を他のテリトリに複写して表示する方法である。もう 1 つは、テリトリを重ねて表示する方法である。

イメージデータの複写は、テリトリ内の任意の矩形領域のイメージで他のテリトリ内の任意の位置を塗りつぶす。矩形領域内の特定のイメージだけを複写することはできない。また、複写するときイメージで塗りつぶすかあるいは AND や OR 等のビット演算を行なうかの指定ができる。モノクロイメージならば OR を使って前景のみを複写することができる。しかし、カラーイメージではカラー同志のビット演算結果の色が不定となるので、前景だけを元の色のまま複写することができない。ビット演算は複写されるイメージと複写するイメージのカラー番号同志で行ない、その結果のカラー番号が合成されたイメージのカラー番号となる。しかし、利用者が直接カラー番号とＲＧＢの値を指定できないので、結果として合成されたイメージのカラーは不定となる。

テリトリを重ねて表示する場合は下に隠れた部分は表示されない。またイメージの複写のような AND や OR などのビット演算はない。

いずれにしても、矩形以外の任意の形状のカラーイメージを重ね合わせて表示することは不可能である。また、同時に使用できるカラー数を越えて色の指定を行なうと表示されるカラーは不定となる。

以上が *YyonX* ver.1.2 におけるカラーの扱いとその問題点である。なお、カラーのＲＧＢの値として各々 0 ～ 65535 までの整数が指定可能である。また、カラー番号とは color クラスのインスタンスを一意に表す数字であり、それ以上の意味はなさない。

```
(make-color &key (:red 0) (:green 0) (:blue 0)) => #<color>

YY Client                Protocol              YY Server
make-color-------------> R G B   ----------->Query&Alloc Color
#<color>  <-------------ColorCode<----------
```

図 1: カラーの作成（ＲＧＢ値を送りカラー番号が返される）

2.2 *YyonX* ver1.3 のためのカラー構造の拡張と定義

任意の形状のイメージを切り貼りして表示するために、利用者がカラー番号とＲＧＢの値の対応を直接指定してカラーを生成する方法が考えられる。しかしこの方法は利用者がカラー番号についてORやANDなどの演算結果を意識してＲＧＢの値を指定する必要があり、使いやすいとはいえない。そこで、背景色を透過する透明色を考えた。２つの矩形領域を重ねたとき、透明色の部分は下に隠れたイメージを表示する。

イメージデータの複写あるいはテリトリの重ね合わせ表示でも透明色の扱いを有効とするために、背景あり／なしのテリトリを定義する。背景ありのテリトリはイメージデータの複写、テリトリの重ね合わせ表示ともに *YyonX* ver1.2 と同じである。背景なしのテリトリは背景が透明色のテリトリである。背景なしテリトリの矩形領域のカラーイメージを背景ありテリトリの上に複写すると透明色の領域は透過して下にあるカラーが有効となる。また背景なしテリトリを背景ありテリトリの上に重ねて表示すると、透明の領域は下のテリトリのカラーが有効となる。さらに背景なしテリトリの表示位置を移動すると自動的に再描画する。したがってカラーイメージの複写、テリトリの重ね合わせ表示ともに任意形状のカラーイメージの表示が可能となる。また、背景なしテリトリ背景ありテリトリともに描画時のカラーに対するビット演算が指定できるがその時のカラーは不定となる。

透明色のＲＧＢの値は全て -1、カラー名は "transparent"、カラー番号は -1 である。

カラーに関する機能の拡張と透明色の導入に対応する、Lisp 関数の作成を行なった。機能拡張並びに新規作成した Lisp 関数を表 1に示す。

2.3 カラーマップ

X Window のカラーの扱いではカラーマップを使う。同じように X Window のクライアントである *Yy* サーバもカラーマップを使用するが、複数のカラーマップを切替えて使用することはできない。そこで、限られたカラー数を有効に活用するために *YyonX* 起動時にカラーマップの扱いを指定できるようにする。

1. １ディスプレイについて１つ（共通）のカラーマップとする。
   YyonX 起動前に X Window で設定されたカラーであっても変更を可能とする。
   これは *YyonX* 起動時にその時点で設定されているカラーを `*all-colors*` にバインドし、必要に応じて `remove-color` や `replace-color` を行えるようにすることである。

2. ルートウィンドウローカルにする。 *Yy* のルートウィンドウにローカルのカラーマップを持ち、それ以下のウィンドウはこれを共通に使う。

2.4 カラーイメージ描画機構

YyonX ver1.3 に試作したカラーイメージ描画機構について述べる。イメージデータはファイル、クライアントの内部データ、サーバのテリトリの３状態に分けられる。カラーイメージはＲＧＢの値を各々４バイトの範囲で保存する。またモノクロイメージは１ビクセルにつき２ビットで保存する。クラ

イアントはイメージデータファイルを読み込みイメージデータオブジェクトを生成する。その際同時に必要なカラーオブジェクトの生成を行なう。表示する場合はイメージデータを *Yγ* プロトコルを通してサーバに送る。またはその逆にサーバからテリトリのイメージをプロトコルを通してクライアント側に取り込み、内部データを生成する。必要に応じてクライアントのイメージデータをファイルに保存することが可能である。(図2)

背景ありテリトリは背景なしテリトリを子供に持つことができる。背景なしテリトリを背景をもつテリトリの上で移動すると、背景なしテリトリに描かれたイメージだけが移動するように見える。また背景ありテリトリのカラーイメージの複写は、その子供の背景なしテリトリのカラーイメージを含めて行なう。(図3)

表 1: 拡張関数一覧

関数名	機能
make-color	RGBからカラーを作る. 作れなければ nil を返す
find-color	RGBからカラーを照会する. なければ nil を返す
make-color-of-name	名前からのカラーを作る. なければ nil を返す
all-colors	グローバル変数. make-color された全てのカラーをバインドする
remove-colors	カラーを解放する
replace-color	カラーのRGBの値を変更する
color-code	カラーオブジェクトのカラー番号を返す
code-color	カラー番号からカラー照会する. なければ nil を返す
query-color	グラフィックストリームのポジションからカラーを求める
query-color-xy	グラフィックストリームのポジションからカラーを求める
initialize-yy	カラーマップの選択を行なう
make-image	イメージオブジェクトを作成する
image-size	イメージの大きさを求める
image-width	イメージの幅を求める
image-height	イメージの高さを求める
image-color	イメージの点のカラーを求める
image-color-xy	イメージの点のカラーを求める
flush-image	イメージオブジェクトを破壊する
save-image	イメージデータを切りとりファイルに保存する
load-image	ファイルから切りとったイメージデータを生成する。
put-image	イメージデータをグラフィックストリームに送る
put-image-xy	イメージデータをグラフィックストリームに送る
get-image	グラフィックストリームからイメージを取り込む
get-image-xy	グラフィックストリームからイメージを取り込む
draw-image	グラフィックストリーム間のイメージを複写する

2.5 効果と性能

イメージ描画を行なうには、*YγonX* ver1.2 では 1 点づつ描画する以外の方法が提供されていなかった。このために *Yγ* Client は 1 点づつ回転や拡大のための行列計算をして位置を求め、テリトリの大きさとの比較を行なう。もし、描画する点の位置がテリトリの大きさを越えていたならば、その点が描画できる大きさにテリトリを拡大する。そして、描画するテリトリと点の位置とカラーをサーバに送る。

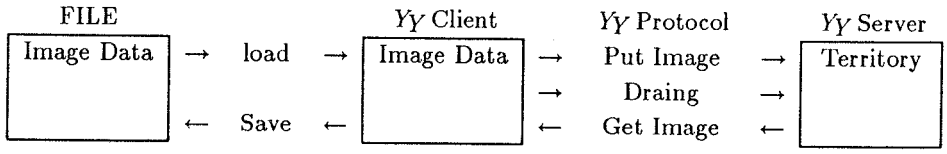


図 2: カラーイメージ描画機構

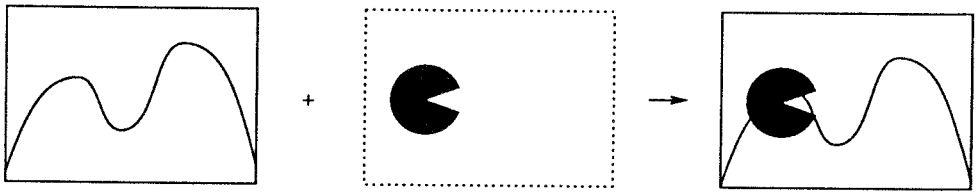


図 3: カラーイメージの重ね合わせ

したがって、描画する点の位置を p、位置の計算に要する時間を f1、テリトリの拡大に要する時間 f2、サーバとの通信に要する時間を f3、サーバが点を描画する時間を f4 とすると、一点を描画するのに要する時間 t は次の式に表せる。

$$t(p) = f1(p) + f2(p) + f3(p) + f4(p) \tag{1}$$

ここで、位置の計算に要する時間とサーバとの通信に要する時間、並びに描画に要する時間は点の位置に関わらず一定である。従ってそれぞれを定数 F1、F3、F4 とすると

$$t(p) = F1 + f2(p) + F3 + F4$$

と書ける。よって複数の点を描画する場合に要する時間は、描画する個数を n とすると次のように表せる。

$$\sum_{i=1}^{n} t(p_i) = \sum_{i=1}^{n} f2(p_i) + n(F1 + F3 + F4) \tag{2}$$

YYonX ver1.3 ではイメージオブジェクトを用いて描画する。また、位置の計算とテリトリの拡大は描画を開始する前に一度だけ行なう。描画するイメージを I として (1) 式にならうと次のようになる。

$$\begin{aligned} t'(I) &= f'1(I) + f'2(I) + f'3(I) + f'4(I) \\ &= f1(p) + f2(p) + f'3(I) + \sum f4(p) \\ &= F1 + f2(p) + f'3(I) + \sum F4 \\ &\quad \{p|p \in I\} \end{aligned} \tag{3}$$

[6] によればパケットの大きさがある範囲内であればサーバとの通信時間はほぼ一定である。この時間は (2) 式の $F3$ に等しい。ここで、通信時間が一定となる範囲内のパケットの最大値を A、パケットの大きさを求める関数を $P(I)$、小数点以下を切り捨てる関数を $int()$ とする。また n 個の pixel からなるイメージを In とすると (3) 式は次のようになる。

$$t'(In) = F1 + f2(p) + int\left(\frac{P(In)}{A}\right) F3 + \sum_{i=1}^{n} F4$$

$$= F1 + f2(p) + int\left(\frac{P(In)}{A}\right)F3 + nF4 \quad (4)$$
$$\{p|p \in I\}$$

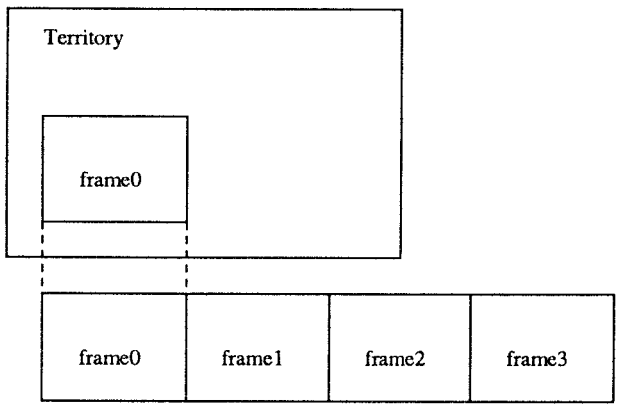
(2) 式と (4) 式を比較すると、ver1.2 の場合はイメージの大きさに比例して描画時間が増大するのに対して、ver1.3 ではメージが大きくなると *Yy* Server の描画時間とほぼ同じような増加をしめすことがわかる。本稿では個々の計測値については使用する機種により異なるので省略する。

3 動画の機構

3.1 フレーム・アニメーション

アニメーションを表現するためには *YyonX* ver1.2 では Bitblt の繰り返しにより実現する以外の方法が提供されていない。すなわち予めビットマップ・ストリームに描画を行ない、表示する矩形領域をワールドと呼ぶ領域に繰り返し Bitblt する。しかしながらこの方法ではクライアントが表示領域をサーバに指示するので、クライアント側で発生するＧＣ等の理由により表示領域を切替える時間間隔が一定にならない。またすでにサーバ側にあるイメージを定期的に切替えて表示するために、クライアントが時間を監視してサーバに Bitblt を指示しなければならい。そこで新たにアニメーションのための機構を追加することにした。動画として認識するために、１枚の絵を表示し次の絵の表示を開始するまでの時間を設定できるように考えた。また、アニメーションの停止、速度変更や巻き戻し、アニメーション中の正しいイベントの処理について考慮した。この機能を実現するするためにフレームとよぶものを考えた。フレームとは同一の大きさのイメージの集合でありそのイメージの１枚１枚についてフレーム番号を持つ。そして、特定の時間間隔でフレーム番号の昇順にあるいは降順にイメージの表示を行なうものである。フレームの機構はテリトリに実現する。フレームは１つのテリトリについて任意の領域に１つのみ設定可能とする。フレームによるアニメーションについて図 4に示す。

利用者は Lisp 関数によってウィンドウ・ストリームにフレームの大きさを指示する。次にフレーム毎にイメージデータを転送する。`run-animation` を実行するとアニメーションを開始する。一度転送されたアニメーションデータは `flush-animation-frame` を実行するまで保存するので、実行と停止が繰り返し行なえる。アニメーション用に作成した関数を表 2にしめす。また、使用例を図 5にしめす。



テリトリにあるフレームのイメージを一定間隔で frame0 から順に frame1、frame2... のイメージに切替えて表示する。

図 4: フレームによるアニメーション

```
(defvar *window-stream* (make-window-stream :left 300 :height 300))

(defun prepare-anime (animation-frame data-file)
  ;; アニメーションデータを animation-frame に転送する
  ...
)

;;; data-file を読み込んでアニメーションを実行する
(defun show-anime (data-file)
  ;; アニメーションフレームの宣言
  (let ((animation-frame (define-animation-frame *window-stream* 0 0
                                                 300 300 50)))
    (prepare-anime animation-frame data-file)
    (run-animation animation-frame)          ; アニメーションの実行
    (flush-animation-frame animation-frame) ; アニメーションフレームの解放
    )
  )
```

図 5: アニメーションの使用例

表 2: フレームアニメーション関数一覧

define-animation-frame graphic-stream left top width height max-frame-number ⇒#<animation-frame> アニメーションフレームを定義する
run-animation animation-frame &key :left :top (:speed 1) (:start-fame-no 0) (:direction :forward) (:presentation-times :endless) ⇒last-frame-number アニメーションを開始しアニメーションと同期して終了する
change-animation-speed animation-frame speed ⇒nil アニメーションの速度を変更する
stop-animation animation-frame ⇒current-frame-number アニメーションを停止する
display-frame animation-frame frame-number コマ送りで表示する
put-frame animation-frame frame-number image &key (:image-position (make-position :x 0 :y 0)) イメージがフレームサイズより大きい場合はフレームサイズでクリップする
flush-animation-frame animation-frame アニメーションフレームの破壊

3.2 フレームアニメーションの性能

フレームアニメーションは *YY* サーバが定期的にイメージを描画するだけであるから、アニメーションを行なうフレームの幅を w 高さを h 描画速度を v とすると、1秒あたりのコマ数 K は次の式に表せる。

$$K = \frac{v}{wh}$$

したがって v が十分速ければ動画として認識できる実行速度が得られる。

[6] に計測したケースでは１０００文字あたりの表示時間が 920ms なので、1文字 7 × 14 ドットとしてイメージの表示速度は $106 pixel/ms$ となる。よってアニメーションフレームの大きさが 100 × 100 pixel であるときの1秒間のコマ数はおよそ 10 コマである。これは例えば映画等が秒２４コマであるのと比べて十分とはいえない。しかし、特別なハードを必要としない簡易なアニメーションシステムとしては実用的であるといえる。

4 終りに

イメージ処理の拡張とアニメーション処理機構の試作を行なった。この結果カラーイメージの加工や再描画が簡単に利用可能となった。ただし、背景なしテリトリによる重ね合わせ処理はサーバでの試作にとどめた。これについてはさまざまな利用方法が考えられるため、有効な活用方法と利用者へのインタフェースを研究中である。現段階ではサーバクライアント間のイメージデータ転送に時間を要するため、実質的に利用できるイメージの大きさに制限がある。また、サーバにイメージデータを持たせるため、利用できるイメージの数や大きさはサーバマシンのメモリ容量に依存する。

今後の課題としては、前述の問題点を解決することと同時に、ディスプレイへの入出力について文字やイメージ、アニメーションの統合を目指すところにある。

参考文献

[1] Masayuki Ida, et. al. : "A Requirement Analysis for a Portable Window System on Top of Common Lisp" IPSJ annual conf. 1990 March

[2] Masayuki Ida, et.al. : An Overview of *YY* and *YYonX* – A CLOS based Window Tool Kit and its Implementation –, Proc. Europal'90, pp245-252 March 1990.

[3] 井田 昌之、他：“*YYonX*: 概要設計”, 情報処理学会第 40 回全国大会, March 1990.

[4] 古坂 孝史、他：“*YYonX* における *YYWS* の設計”, 情報処理学会第 40 回全国大会, March 1990.

[5] 田中 啓介、他：“*YYonX* における *YY-server* の設計”, 情報処理学会第 40 回全国大会, March 1990.

[6] 古坂 孝史、他：“*YYonX* 実行モデル (2)”, 情報処理学会第 41 回全国大会, Sept. 1990.