オーディオビジュアル複合情報処理 28－3
（2000．3．3）

Java版BMLブラウザ描画部高速化に関する検討

宮澤 敏記[1)] 亀山 渉[2)] 富永 英義[1),2)]

[1)] 早稲田大学 理工学部 電子・情報通信学科
〒169-8555 東京都新宿区大久保3-4-1
miyazawa@tom.comm.waseda.ac.jp

[2)] 早稲田大学 国際情報通信研究センター
〒169-0051 東京都新宿区西早稲田1-21-1 早大西早稲田ビル5階

あらまし

本稿では，JavaによりBML対応ブラウザを作成する中で実行速度の遅い部分が生じたため，Javaに特化したアルゴリズムを考案し改善を図ったので報告する．考案したのは，グレースケールフォント(階調フォント)作成アルゴリズム及び半透明合成を含む一部オブジェクト変更に伴う再描画アルゴリズムである．グレースケールフォントに関しては，アルゴリズムを複数考案し実行時間を比較した上で，ブラウザに組み込むアルゴリズムを決定した．一方考案する再描画アルゴリズムでは，階層的に配置されたオブジェクトの一部を変更する場合，再帰的に描画領域の重なり判定を行なうことでオブジェクトの数・位置・種類に依存せず，かつ描画面積を最小にすることが可能となった．

キーワード Java，グレースケールフォント，再描画，アルゴリズム，BMLブラウザ

A Study on Graphics Acceleration Algorithms for Java version BML Browser

Toshinori MIYAZAWA[1)] Wataru KAMEYAMA[2)] Hideyoshi TOMINAGA[1),2)]

[1)] Dept. of Electronics, Information and Communication Engineering, WASEDA University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555 JAPAN
miyazawa@tom.comm.waseda.ac.jp

[2)] Dept. of Global Information and Telecommunication Institute , WASEDA University
Nishi-Waseda Bldg. 5th floor 1-21-1 Nishi-Waseda, Shinjuku-ku, Tokyo, 169-0051 JAPAN

abstract:

In this paper, we propose graphics acceleration algorithms for Java version BML Browser. During the process of developing BML Browser using Java, there were slow parts in the Rendering Engine of BML Browser, which were grayscale decoding and repainting method including alpha composition. For grayscale decoder, we propose a few algorithm and compare running time, and we decide algorithm to be used in BML Browser. For repainting method, we created a process which examines reflexively whether repainted object overlaps(is overlapped by) other objects disposed in layer. This process doesn't depend on number of objects, position of object or object's type and enables repainting method to draw the smallest area.

keyword Java, grayscale font, repainting method, algorithm, BML Browser

1. はじめに

日本のデータ放送におけるマルチメディア符号化方式は電波産業会によりXMLをベースとしたBML方式に定められ，2000年12月にBSデジタル放送が開始される．それに伴い，我々はBML方式対応ブラウザをJavaにより開発することとなった．開発言語にJavaが選ばれた理由は，将来的にSTBへの組み込みを考慮したためである．Javaの利点には，以下がある．

- 共通プラットフォーム
  多種多用なCPUやOS環境が想定されるSTBにおいても，アプリケーションに互換性がある
- ネットワークダウンロード型アプリケーション
  ソフトウェアのバージョンアップなどにJavaアプレットが有効である．
- Java言語の有用性
  ポインタの無い言語使用はデバッグの労力が減る．またオブジェクト指向デザインによりソフトウェアに再利用性がある

本稿では，BMLブラウザ描画部(Rendering Engine)における実行時間の遅い部分をJavaに特化したアルゴリズムを用いて改善を行なった．

2. BMLブラウザの構造

BMLブラウザの構造は図1のようになっている[2]．

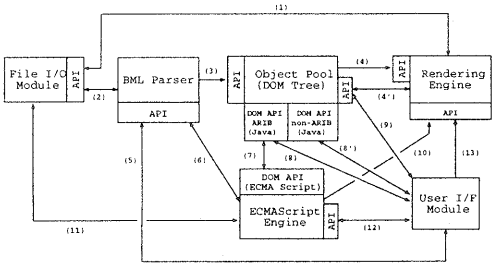

図 1: ブラウザの構造

主な動作は，File I/OモジュールからBML文書がParserモジュールに渡され，解析結果がObject Poolモジュールに保持される．Object PoolからRendering Engineモジュールに描画指示が出る．画像などのコンテントデータはParserとは別に(1)のルートを通りRendering Engineに渡される．改良を行なったのは，Rendering Engine内に含まれるグレースケールフォントデコーダ及び再描画処理機能である．

3. グレースケールフォント

グレースケールフォント（階調フォント）とはフォントの輪郭をフォントとは異なる色で縁取りしたものである(図2)．用いられる理由は，受信機は低価格で解像度が低い(モニタから離れて視聴)，フリッカが生じる，フォント色と背景の色が近い場合があるなどによりフォントの輪郭を強調し視覚し易くする必要があるためである．

図 2: グレースケールフォント

STB(Set Top Box)に近いスペック[1]の環境においてコンテントデコーダ部の性能[2]は以下であった．

表 1: コンテントデコーダ性能

JPEG	テキストのみ	テキスト+グレースケール
220[msec]	2300[msec]	62400[msec]

この場合，グレースケールフォント（テキスト+グレースケール）はテキストのみに比べて約30倍実行時間がかかるため，この部分を改良する必要があった．

3.1 改良前のグレースケール作成方法

- 左上点から右下点まで走査し，フォント色を検出
- その画素周辺の画素を全て縁の色で塗りつぶす
- 最後に最初のフォントを重ね合わせる

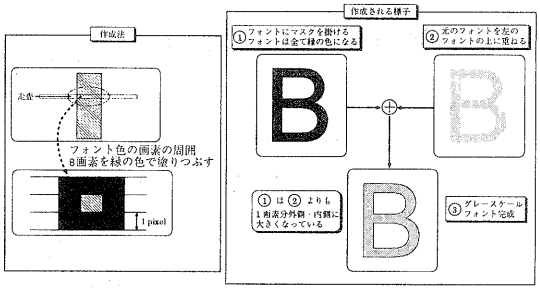

図 3: 改良前のグレースケール作成法

この方法では，実行速度は走査時間に依存する．輪郭を1階調増やすごとに，1回画像全体を走査する必要がある．ゆえに理論的には，階調数nに対して実行時間は$o(n)$のオーダであることが分かる．

3.2 輪郭からグレースケールを作成する方法

- フォントの輪郭を作成
- 輪郭内をフォント色で塗りつぶす(図4右上)
- 輪郭を中心として特定の太さを持った線を縁の色で描く(図4左上)

[1]CPU:MMX Pentium 166MHz, RAM:96MB, Video RAM:1.1MB(TOSHIBA LiberettoSS 1000

[2]JPEG: 画像サイズ(260x125 画素)，テキスト(146 文字)，グレースケール(4 階調)

・ 図 4の左上図の上に右上図を描画する
・ フォントを上にして，重ね合わせる．(図 4下)

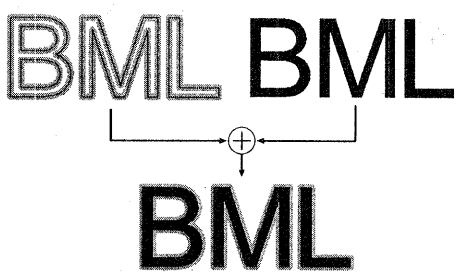

図 4: 輪郭からグレースケールの生成

この場合，階調数が 1 増えると，太さを持った輪郭の作成及びその輪郭線の描画処理が加わる．ゆえに理論的には，階調数 n に対して実行時間は $o(n)$ のオーダであることが分かる．

3.3 フォントを複数重ね合わせる方法

・ フォントを縦横に 1 画素ずつずらし，縁の色で描画
・ 最初のフォントを重ねる

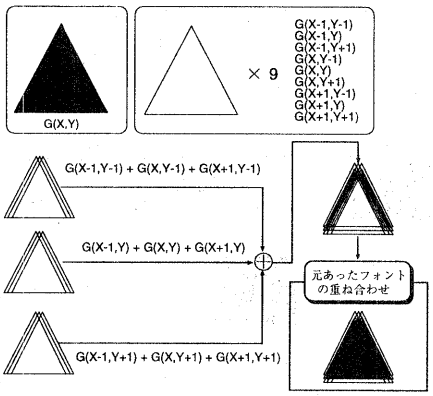

図 5: フォント重ね合わせ法

2 段以上の階調数へは，さらに外側まで 1 画素ずらすことで対応可能である（2 段目は 5×5 回描画）．この手法では実行速度はフォントをずらして描画する回数に依存する．階調番号 k に対して $(k+2)^2$ 回フォントをずらす必要があるので，階調数 n に対して総描画回数は $1+\sum_{k=1}^{n}(2k+1)^2$ である．

理論的には"階調数nー (グレースケール実行時間)' の関係は"nー $(1+\sum_{k=1}^{n}(2k+1)^2)$" の関係に類似すると考えられる ($o(n^3)$ のオーダ)．

3.4 各手法の比較

図 6の結果より，改良した両アルゴリズムはともに従来の方式よりも実行時間が改善されている．'重ね合わせ法' の実行時間は理論値とは異なり，階段状の結果となり，3 段目と 4 段目の間で' 重ね合わせ法' と' 輪郭からの生成法' の実行時間が逆転している．ARIB の運用では階調数は 4 段までである，そのため実行速度の観点から，階段状になる原因を考察する必要が生じた．

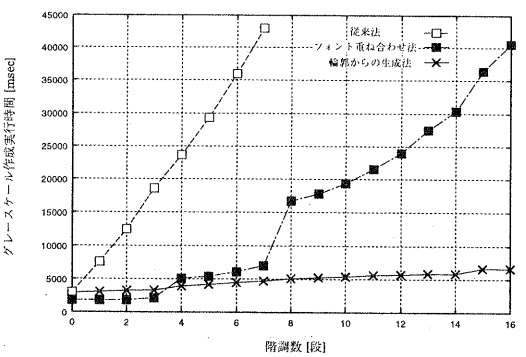

図 6: グレースケールフォント描画実行時間比較

3.5 重ね合わせ法の改良

実行時間が階段状になる原因は，Java VM による自動的なメモリ解放機能 (ガベージコレクション) によることが判明した．具体的には，フォントをずらす毎に新しいフォントオブジェクトを生成しているため，メモリ領域を多量に使用してしまう．ゆえに，フォントをずらす回数を減らすことが可能ならば，メモリ解放による実行時間の増加を抑えることができる．図 7に改良を行なった様子を示す．この図では 1 つのフォントの代表点の軌跡を表したもので，例えば 2 段目のずらしたフォント群というのは内側から 2 番目のフォントの輪郭を作成するためにずらしたフォントの軌跡である．2 段目のフォントの代表点と 1 段目のフォントの代表点が一致するということは，完全に一致するフォントを書き直しているにすぎないため無駄である．ゆえに改良後の方式では，フォントの軌跡をリング状にし，完全に一致するフォントをなくすことで描画回数を抑えた．改良後の描画回数は $1+8\sum_{k=1}^{n}k$ であり，$o(n^2)$ のオーダとなる．階調数が 4 段である場合，描画回数を 165 回から 81 回に減らすことができる．実行時間の結果は図 8のようになる．この手法を現在 BML ブラウザ内に組み込んでいる．

4. 一部オブジェクト変更に伴う再描画

BML ブラウザでは図 9のようにオブジェクトを描画している．オブジェクトは長方形 (以下ウィンドウと書く) であり，種類は不透明色のみである場合と半透明色を持つ場合がある．半透明色が含まれる場合は以下に述べる合成 (α ブレンディング) が必要である．半透明色を考慮した再描画法を考案し，最小の描画面積を実現することを目的とする．

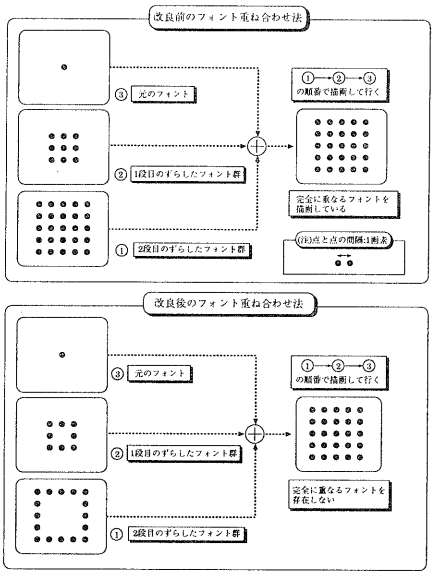

図 7: フォント代表点の軌跡

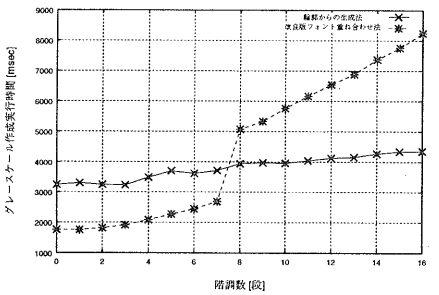

図 8: 改良版フォント重ね合わせ法実行時間

4.1 αブレンディング

不透明画像に半透明画像を重ね合わせる場合，画素を不透明:(r,g,b), 半透明:(R,G,B, α) とすると α 合成は次の式 (r のみ示す) による．

$$r_{new} = r \times (1 - \alpha) + R \times \alpha \qquad (1)$$

不透明画像に対し，半透明画像が 2 つある場合，画素 A1($(R_1, G_1, B_1, \alpha_1)$, 画素 A2$(R_2, G_2, B_2, \alpha_2)$) は奥にある画素から α 合成を行なわなければならない．なぜならば，A1,A2 の順で式 (1)に代入すると次式となり，

$$r_{new} = r - r(\alpha_1 + \alpha_2) + R_1\alpha_1 + R_2\alpha_2 + (r - R_1)\alpha_1\alpha_2 \qquad (2)$$

$(r - R_1)$ の項により R_1 と R_2, α_1 と α_2 を入れ換えた場合 (A2,A1 の順)，r_{new} の値は異なるからである．

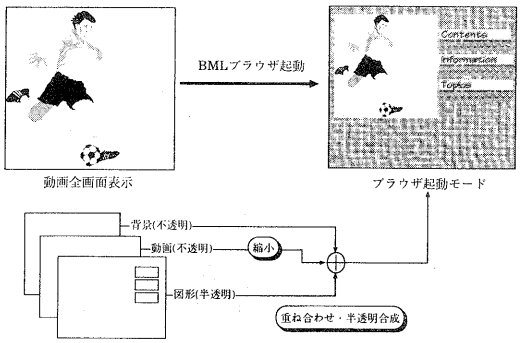

図 9: オブジェクト描画例

4.2 提案する再描画法

- 再描画するウィンドウの上位 (手前側) に不透明なウィンドウがあるかどうか調べる
- ある場合重なるか判定
- 重なる場合，重なる部分を取り除く (図 10左)
- 以後重なる部分を除かれた再描画ウィンドウの上位に半透明ウインドウがあるか調べる
- ある場合，重なり判定し，重なる場合，重なりを再描画ウィンドウに足して行く (図 10右)
- 再描画ウィンドウに α 成分が含まれる場合，別の処理が必要である (後述)

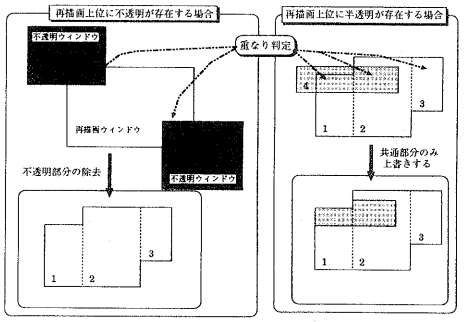

図 10: 重なり除去及び重なり合成

以下に重なり判定法，再描画ウィンドウに α 成分が含まれる場合の処理を説明する．

4.3 重なり除去再描画ウィンドウの作成法

4.3.1 重なり領域判定法

二つのウィンドウがある場合，下位ウィンドウの取りうる形には図 11がある．下位ウィンドウの形状は，上位・下位ウィンドウのそれぞれ左端・右端頂点座標を用いて，if 文 5 回で pattern A ～ P と重ならない場合のうちのどれか一つに特定できる．重なり pattern 特定のための分岐処理は図 12のようになる．

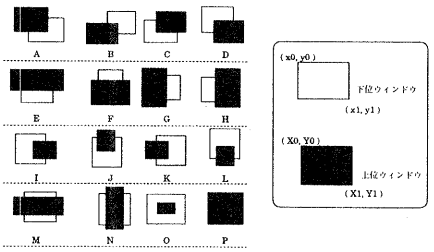

図 11: 下位ウィンドウの形状分類

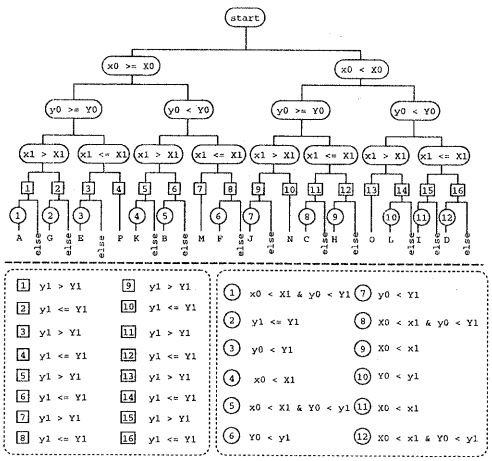

図 12: 二分木による判定

4.3.2 ウィンドウ分割

図10では，再描画ウィンドウの上位に不透明ウィンドウがある．この場合，再描画ウィンドウは3つに分割されている．ウィンドウを分割する理由は，分割された各々のウィンドウに対し重なり領域判定法が再利用可能であるためである．例えば，重なり領域判定で pattern A，O が選択された場合，図13のように分割する (分割は縦方向に統一した)．この場合，分割された2つのウィンドウは，Z インデックス（階層番号）の等しいウィンドウが複数存在するとして処理する．以後，重なり領域判定，ウィンドウ分割を繰り返せば図10左側に示した重なりが除去された複数のウィンドウが作成される．

4.4 再描画ウィンドウに α 成分が含まれる場合

再描画ウィンドウの下が透けて見えるため，奥にあるウィンドウも描画する必要がある．半透明ウィンドウは奥にある画像から描画する必要があるので，再描画ウィンドウを描画する前に，次の処理が必要である．

- 分割された各再描画ウィンドウに対し以下を行なう
- 再描画ウィンドウと全ての下位ウィンドウに対し AND 領域をとる (図 14)

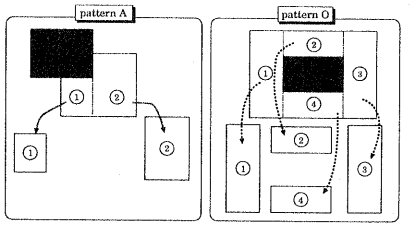

図 13: ウィンドウ分割例

- AND 領域をとった下位ウィンドウは奥側から描画する．その際，上位の AND 領域に不透明ウィンドウがあれば重なり判定を行ない，見える部分のみ描画し，上位が半透明であるならばそのまま描画する

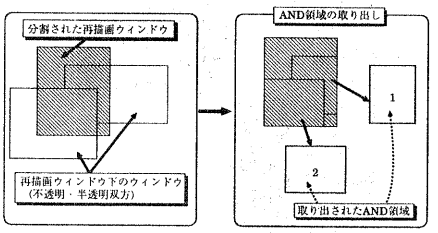

図 14: 再描画ウィンドウとの AND 領域

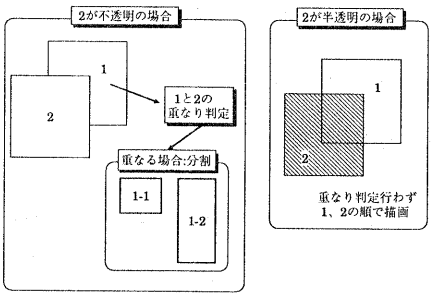

図 15: 再描画ウィンドウ下での重なり判定

5. Java によるコーディングでの考慮点

Java では setClip() メソッドで領域を指定することで，Clip 領域のみ描画される．以上で扱ったウィンドウは，ウィンドウの Z インデックス・座標・種類 (不透明 or 半透明) を保持するオブジェクトとして扱う．ウィンドウは幾つに分割されるかは実行後でなければ分からないので，要素数を増減可能である Vector 型配列にウィンドウオブジェクトを格納している．以下の場合に対しそれぞれフラグを立て，重なり判定を試みる回数を減らすことで高速化を図った．

- 再描画ウィンドウが半透明である場合
- 再描画上位に不透明ウィンドウが存在する場合
- 再描画上位に半透明ウィンドウが存在する場合
- 再描画下位に不透明ウィンドウが存在する場合
- 再描画下位に半透明ウィンドウが存在する場合

6. 再描画実行例

再描画実行例を図16に示す．この例では，再描画ウィンドウが不透明で，上位に不透明ウィンドウがあるため重なり除去し，その後，α合成している．白く刳り貫かれている部分が，再描画ウィンドウの上位に不透明ウィンドウがあると判断し描画していないことを示している．一方，半透明ウィンドウは再描画ウィンドウの外側まで存在しているが再描画ウィンドウ領域のみ描画されている（再描画ウィンドウで濃さが異なっている部分）．このようにこのアルゴリズムでは再描画ウィンドウの形を柔軟に設定できることが分かる．

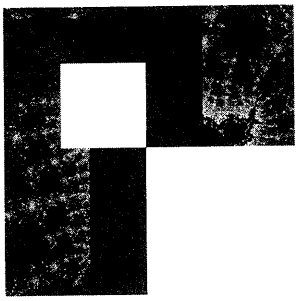

図 16: 再描画実行例

7. 再描画アルゴリズムの評価

7.1 重なり除去処理時間

図17に示した場合について，重なり除去処理時間を計測する．画像サイズは500x500 画素で，縦横 50 画素ずつずらしていく．全て描画した場合と重なり判定した後に再描画ウィンドウを描画した場合を比較した．

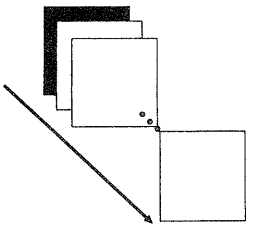

図 17: 重なり除去処理時間評価

図18より現実的なウィンドウ数では重なり除去時間は影響しないことが分かる．なお，ウィンドウ数を増やした場合の描画時間は表2ようになる．

表 2: 重なり除去処理時間

ウィンドウ数	100	1000	5000	10000
重なり処理時間 [msec]	710	770	820	930

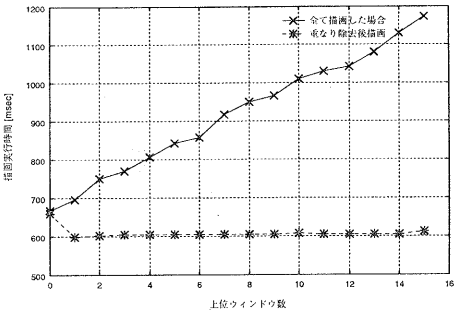

図 18: 再描画実行時間

7.2 再描画ウィンドウにα成分含む場合

図19における 3 番目の画像を再描画する．全て描画する場合と提案方式の場合の描画時間を比較する．不透明・半透明ウィンドウが混在するように設定した．

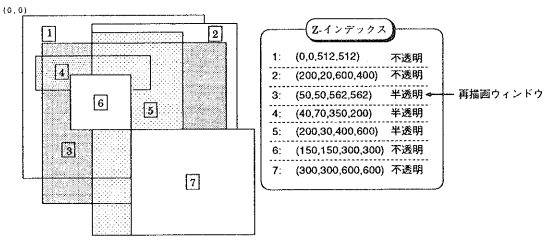

図 19: 再描画ウィンドウにα成分が含まれる場合

表 3: 描画時間比較

全ウィンドウ描画	1210[msec]
提案方式	1100[msec]

8. まとめ

グレースケール作成アルゴリズムは，複数のフォントを 1 画素ずつずらし描画する中で完全に一致するフォントは描画しない'改良版フォント重ね合わせ法'が現状で最速であるため，BML ブラウザに組み込んだ．一方，最小描画面積で再描画可能なアルゴリズムを考案することができた．なお，このアルゴリズムはオブジェクトの移動や拡大縮小などのアニメーションにも応用可能である．変化前と変化後の各オブジェクトへ重なり判定を行ない，重ならない領域を空領域として描画すれば良い．

参考文献

(1) 電波産業会， STD-B24 “データ放送符号化方式と伝送方式”,1999

(2) 亀山渉, 尾崎誠司, 目々澤健治,BML Browser の構成方法と実装に関する検討, 信学春全国大会 (2000)