汎用動画像処理ソフトウェアライブラリ – MAlib –

Generic Video Image Processing Library – MAlib –

飯尾 淳* 谷田部 智之* 比屋根 一雄*
iiojun@mri.co.jp tyatabe@mri.co.jp hiya@mri.co.jp

概要 PC の高性能化、デジタルビデオやデジタルカメラの普及、およびそれらの機器と PC とのインタフェースの進化に伴い、画像情報の PC を用いた加工や蓄積は現在とても身近な技術となりつつある。そのような背景のもと、動画像処理を行なうソフトウェアの開発は、ニーズの高い分野として注目を浴びている。しかし現在のところ手軽に利用できる汎用的なソフトウェアライブラリが存在していないため、個別に開発が重ねられているといった問題が残されている。そこで、それらの動画像処理アプリケーション開発の促進を目的として、汎用に利用することができる動画像処理ソフトウェアライブラリを構築し、LGPL ライセンスのもとでオープンソースソフトウェアとして公開した。

1 はじめに

近年の PC の性能向上は著しく、データ量の多い動画像を高速に処理することが可能になりつつある。またデジタルビデオやデジタルカメラの普及、およびそれらの機器と PC とのインタフェースの進化に伴い、画像情報の PC での加工や蓄積は既に身近な技術となっている。静止画像の処理ソフトウェアは現在様々なツールやライブラリが存在し、画像処理技術普及の牽引役を担っている。画像の認識処理についても、静止画像を対象にしたものは古くから開発用ライブラリ [1, 2] が存在し広く利用されてきた。また imlib や gdk-pixbuf など、静止画像に関してはオープンソースの画像処理ライブラリも普及しており、自らのアプリケーションに簡単に組み込むことができるようになっている。

ところが映像処理・動画像処理のソフトウェアに関しては、ビデオキャプチャや IEEE1394 を用いたデジタルビデオ接続を対象としたビデオ編集アプリケーションなど、個別のメーカによる商用アプリケーションは存在するものの、手軽に利用できる映像利用アプリケーション開発ライブラリで汎用的なものは存在しない。映像認識に特化したライブラリとしては、インテルの OpenCV[3] やマイクロソフトの VisionSDK[4] などいくつかのライブラリが存在するが、本プロジェクトの目的は、画像だけでなく音声も含めたデータのハンドリングや映像データの入出力まで含めたより汎用的なライブラリの開発にある点で、これらのライブラリとはややスコープが異なる。

今回報告する汎用動画像処理ライブラリの開発は、ImageMagick[5] や Netpbm[6] に類した形式の、映像データを対象とするツールの構築を当面の目標に掲げて始められた。現時点ではまず第一段階として、それらのツールだけでなく他のアプリケーションに容易に組み込むことが可能な汎用動画像処理ライブラリとしてのフレームワークの構築が終了したところである。

本論文では、今回開発した汎用動画像処理ライブラリの概要、基本アーキテクチャと実装の概要、ライブラリの応用例として作成した移動物体認識アプリケーションの紹介について述べる。また今後の計画を述べるにあたり、企業におけるオープンソース開発事例のひとつとしての側面から捉えた本プロジェクトの立場について最後に補足する。

2 汎用動画像処理ライブラリ

汎用の動画像処理ライブラリ提供の目的から、とくにプラットフォームは限定せず移植性の高いオープンソースライブラリとして開発することに留意したが、まず開発のプラットフォームとして Linux を選択した。その理由としては、本開発の

*株式会社 三菱総合研究所 情報通信研究部

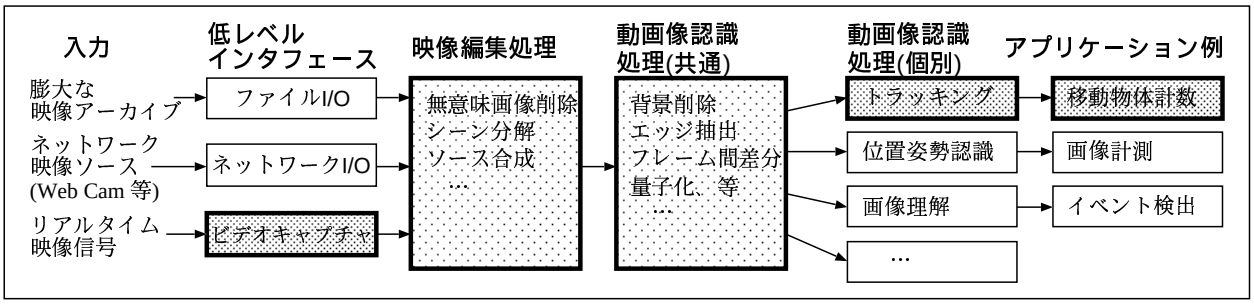

図 1: MAlib を利用した動画像処理アプリケーション

ベースとなったプロジェクトが Linux 用の画像処理ソフトウェア [7] であったこと、Linux を基盤とした場合、リアルタイムシステム、組込み機器から高性能ワークステーションまでのスケーラビリティが望めることなどが挙げられる。とくに組込み機器への展開については、現状では本成果を適用するまでには至っていないが、組込み用ハードウェアの更なる高性能化が期待されているところでもあり、今後目を離せない分野のひとつと考えている。

さて本プロジェクトで構築し、公開したライブラリ MAlib は、動画像処理に関する汎用のデータ構造とプログラムを C 言語のライブラリ形式で提供する。その名称は、以下の目的を含む汎用的なライブラリの提供という、本プロジェクトが目指す最終的なゴールに由来する。MAlib は、以下の目的に合致するように汎用的に利用されるライブラリとして位置づけられる。

- 動画像認識、動画像を用いた信号処理等、動画像メディアを中心とした信号解析、認識処理の基礎を提供するライブラリとなる (Media Analysis)
- さらに動画像である特性を活かし、動画像中に含まれる移動物体の解析に特化した処理を提供する (Motion Analizing)
- 認識処理、解析処理の基盤となる一方で、認識・解析処理を行なうための素材作成などに利用できるような、動画像 (映像) 処理基盤を提供する (Movie Architecture)
- さらに音声との同期などを考慮し、映像オーサリングツールを構成する要素としてのライブラリとしても利用可能とする (Media Authoring)

本プロジェクトの最終的なゴールは、あらゆる動画像処理に適用できるライブラリの構築である。まずその第一歩として、MAlib は Linux プラットフォーム上での動画像処理の基盤を提供する。具体的には、Video4Linux を利用したリアルタイム動画像認識処理アプリケーション作成者、ファイルからの入力を想定した複数の動画像を扱うアプリケーションの作成者による利用を想定し、それらのアプリケーションの作成を支援するソフトウェア部品としてのライブラリを提供する (図 1)。図 1 において網掛けの範囲で示している箇所が、現在の実装でライブラリが提供する機能を示している。

3 オブジェクト指向による実装

MAlib は、組込み機器への適用などの汎用性を考慮して C 言語で記述されているが、オブジェクト指向の概念に従って設計されている。C 言語でオブジェクト指向的記述を行なうことの意味は、汎用性と保守性の両立にある。すなわち、C 言語で実装することにより汎用性を高めるとともに、構造体と関数ポインタを利用してオブジェクト指向的に記述することにより、ライブラリ自身のメンテナンス性を向上させることが可能となる。

MAlib におけるオブジェクト指向的な実装は、GTK+[8] でのオブジェクト指向の記述に倣い実現した。ただしシグナルなどの概念が不要であるため、若干簡略化された実装方法となっている。ここではまず、MAlib のクラス階層を示す (「*印」がついているものは今後拡張する予定)。

```
MalibObject (全てのルートクラス)
  +- MalibHolder
  |     +- MalibGtkDisplay
  |     +- MalibFileOutput (*)
  |     +- MalibNetworkOutput (*)
  |     +- ...
  |     +- MalibBuffer
  |          +- MalibPlainBuf
```

```
|          +- MalibRingBuf
|          +- MalibLineBuf
+- MalibFrame
|    +- MalibFrameAV (*)
+- MalibSource
     +- MalibBttv
     +- MalibMpegFile
     +- ...
     +- MalibFilter
          +- MalibDelay
          +- MalibFrameDiff
          +- MalibGenericFilter
          +- MalibGrey2Bw
          +- MalibNegative
          +- MalibRgb2Grey
          +- MalibRgb2Yuv
          +- MalibSepia
          +- MalibMovingAve
          +- MalibSpatial3x3
          +- MalibMerger
               +- MalibOverlap
               +- ...
```

以上に示したとおり、全てのクラスは抽象クラス `MalibObject` を継承して実現される。また次に述べる Source、Holder、Filter、Buffer に相当する `MalibSource` などのクラスは全て抽象クラスである。

各クラスにおけるメンバ関数の実装時には、各インスタンスを可能な限り上位のクラスのオブジェクトとして取扱うように記述することに留意した。すなわち、後述する MAlib のアーキテクチャとして「Filter は Buffer を入力にとり、Holder は Source を入力にとる」といった基本的な概念があるが、それらのハンドリングにおいてはその対象が `MalibBttv` クラスのインスタンスであったとしても `MalibSource` 型として取扱う。各メンバ関数の実装においては、このように抽象度を高めた記述とすることを心掛けた。

4　基本アーキテクチャ

動画像処理の流れは、1. 映像ソースからバッファへの動画像データの入力、2. バッファから読み出したデータに対する処理を行なって新たなバッファへ結果を格納する、という組み合わせを基本とし、複雑な処理を行なう場合はそれらの処理を多段階に繋ぎ合わせることで実現する (図 2)。

入力された画像データは基本的には各フレームを並べたものとなるが、映像フレームをリングバッファ上に並べたリングフレームアーキテクチャ(図 3) を用意することで、永続的な動画像処理を可能とした。なお時間差分の処理など過去の

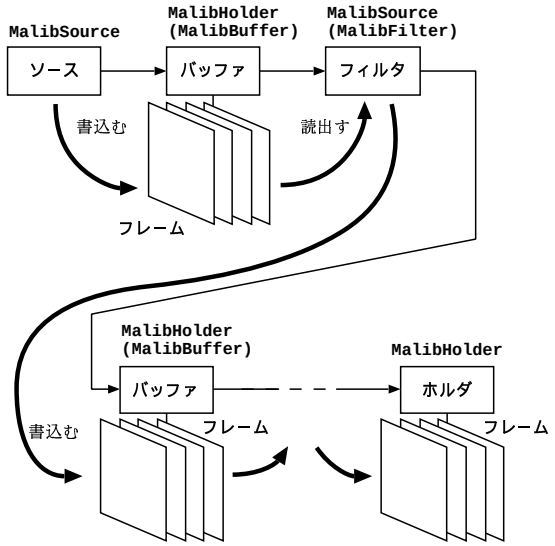

図 2: 動画像処理の流れ

フレームまで遡って画像処理を行なう必要がある操作の場合に対してはこのリングバッファの利用が基本となるが、各時点でのフレームのみを対象としフレームデータの保持が不必要な操作のために単フレームのバッファも用意している。

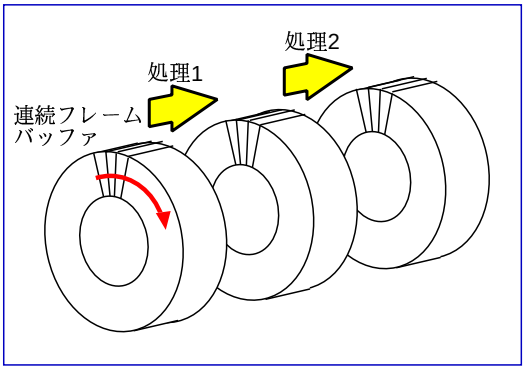

図 3: リングフレームアーキテクチャ

Malib の設計においては、前節のクラス階層の紹介したとおり、まずこのフレームワークにおける基本的な概念として Source と Holder、およびそれらのサブクラスとして定義される Filter と Buffer のクラスを導入した。

Source は映像データのソースとなるクラスであり、Holder は映像フレームのデータを保持するクラスである。Holder の基本的な機能は、Source を入力にとりデータを保持することである。Filter の機能は何らかの処理を行なって後段のバッファへ加工後のデータを渡すことであるから、Filter に接続された Holder からみると Filter 自身が映像ソースとなる。したがって Filter は Source のサ

ブクラスとして定義することが妥当である。Filter で拡張される重要な機能は、Filter への入力がある、ということが挙げられる。

Source が入力の端点となる一方で、Holder は出力側の端点としてとらえることができる。Holder のサブクラスとして定義される Buffer は、Filter へ接続して後段へデータを渡すことができる点が拡張されている。

これらのクラスは次のように接続され、実際の処理が行なわれる。

$$\text{Source} \rightarrow \text{Buffer} \rightarrow \text{Filter}$$
$$(\rightarrow \text{Buffer} \rightarrow \text{Filter} \cdots) \rightarrow \text{Holder}$$

「(→ Buffer → Filter ···)」は処理が多段階になり得ることを示している。なお単純な一連の処理だけではなく、複数の入力を持つクラスや複数のバッファへの処理結果の分配なども可能としているため、一般には複雑な有向グラフを形成することになる (図 4)。なお前述の例や図 4 における矢印は処理の流れを示すものであり、実際のポインタ参照は図上の矢印の逆となることに注意。

なお現時点では、表 1 に示す 11 種類のフィルタが用意されている (記載順序は前述のクラス構成の順番に準じた)。

表 1: MAlib が提供するフィルタ群

MalibDelay	時間平滑化フィルタ
MalibFrameDiff	時間差分フィルタ
MalibGenericFilter	汎用フィルタ
MalibGrey2Bw	グレー→白黒コンバータ
MalibNegative	色調反転フィルタ
MalibRgb2Grey	RGB →グレーコンバータ
MalibRgb2Yuv	RGB → YUV コンバータ
MalibSepia	セピア色フィルタ
MalibMovingAve	3 × 3 × 3 時空間移動平均フィルタ
MalibSpatial3x3	汎用 3 × 3 空間フィルタ
MalibeOverlap	重ね合わせフィルタ

MAlib の利用者が新たなフィルタを作成する方法はふたつある。ひとつは、これらを継承して新たなフィルタクラスを実装する方法である。また、MAlib が提供するデータ構造に対応した画像処理アルゴリズムだけを実装した関数を記述し、汎用フィルタのパラメータとして関数ポインタを与える方法も用意されている。後者を利用することにより、新たなフィルタを容易にテストすることもできるようになっている。

5 フィルタの実装例

フィルタの実装の例として、色調反転処理フィルタ MalibNegative を記述したソースファイル negative.c を示す[1](図 5)。実際のクラス構造の定義は、対応するヘッダファイル negative.h で行なわれる。

```
#include "negative.h"

/* private function prototypes ***********************************/
static void     malib_negative_write_frame_data (MalibNegative* filter,
                                                 MalibFrame* frame);

/* virtual function table ****************************************/
static MalibNegativeClass malib_negative_class =
{
  /* it is ok to use 'malib_filter_delete' instead of preparing
     'malib_negative_delete, because MalibNegative has
     no additional data than MalibFilter */
  (void (*)(MalibObject*))              malib_filter_delete,
  (void (*)(MalibSource*, MalibFrame*))  malib_negative_write_frame_data
};

/* public functions **********************************************/
MalibNegative*
malib_negative_new ()
{
  MALIB_FILTER_GENERIC_NEW ( MalibNegative, &malib_negative_class,
                             MALIB_FRAME_COLORMODEL_RGB
                             | MALIB_FRAME_COLORMODEL_GREY );
}

MalibNegative*
malib_negative_new_with_buf (MalibBuffer* buf)
{
  MALIB_FILTER_GENERIC_NEW_WITH_BUF ( MalibNegative, malib_negative_new,
                                      malib_negative_set_buffer, buf );
}

void
malib_negative_set_buffer (MalibNegative* filter, MalibBuffer* buf)
{
  g_return_if_fail (filter && buf && ((MalibHolder*)buf)->frames[0]);

  /* set the maximum value of this filter */
  filter->max_val
        = (1 << (malib_buffer_get_current_frame(buf) ->depth)) - 1;

  malib_filter_set_buffer ((MalibFilter*)filter, buf);
}

/* private functions *********************************************/
static void
malib_negative_write_frame_data (MalibNegative* filter, MalibFrame* frame)
{
  MalibBuffer* input;
  int i, max_val;
  unsigned int image_size;

  /* the flag whether we need to propagate previous section */
  int need_increment = 0;

  /* pointers to previous frame data and data area to store
     the result of calculation */
  int* from;
  int* to;

  g_return_if_fail (filter && frame);
  g_return_if_fail (((MalibFilter*)filter)->buf
                    && frame->data && filter->max_val);

  input = ((MalibFilter*)filter)->buf;
  to    = frame->data;

  max_val = filter->max_val;

  /* increment previous buffer data */
  MALIB_OBJECT_COUNT_REFERENCES (filter, need_increment);
  if (need_increment)
    {
      malib_holder_increment_frame ((MalibHolder*)input);
    }

  /* get pointers to rgb frame data */
  from = malib_buffer_get_current_frame (input) ->data;

  image_size = malib_filter_calc_output_image_size ((MalibFilter*) filter);

  /* calculate greyscale value of rgb data */
  for (i = 0; i < image_size; i++)
    {
      *to++ = max_val - *from++;
    }
}
```

図 5: フィルタ実装の例 (negative.c)

上記フィルタのソースコードは、プライベート関数のプロトタイプ宣言、バーチャル関数テーブルの定義、パブリック関数定義 (コンストラクタほか)、および実際のフィルタ処理を行なう関数定義から構成される。

本クラスでは、実際のフィルタ処理はバーチャ

[1] ライセンス表示は省略。

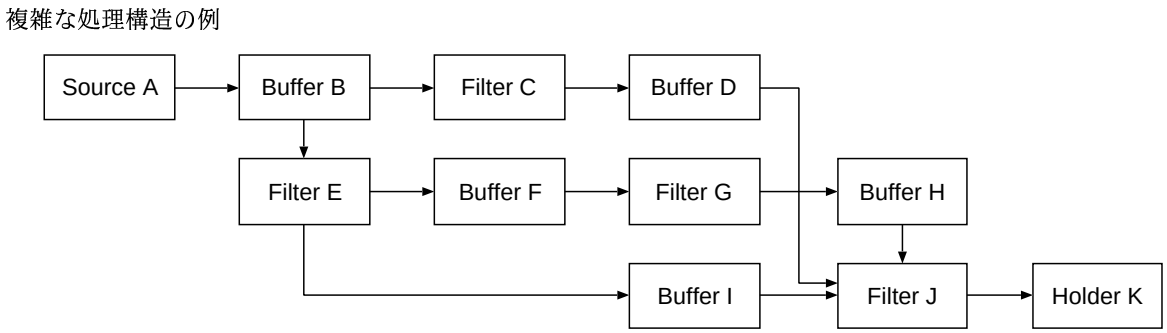

図 4: Source - Filter - Buffer - Holder のリンク構造

ル関数として定義されるため、ファイルスコープに閉じたプライベート関数として定義されている。この関数のエントリポイントは関数ポインタとしてバーチャル関数テーブルに格納されており、実際には上位の抽象クラス `MalibFilter` で用意されるバーチャル関数の実体としてアクセスされる。

ここではフィルタ定義の概略を紹介するに留める。各種マクロ定義など本ソースの詳細については、公開されているソースコード一式を入手し、そちらを参照していただきたい。

6 リファレンスカウンタの利用

MAlib における各オブジェクトは、リファレンスカウンタを持つ。オブジェクトのリファレンスカウンタは、現在のところ以下の目的で利用されている。

- リンクを辿ったオブジェクトの自動的消去
- 画像フレームの更新
- フレーム情報の共有管理

本節では、とくに重要なオブジェクトの消去処理および画像フレームの更新処理について詳しく述べる。フレーム情報の共有管理は、各フィルタが持つ対象フレーム情報の変更を行なう際に、共有されている参照数を管理するためリファレンスカウンタを利用するものである。

不要になったリンク構造を成す画像処理オブジェクトの消去は、関連するリンクを全て辿って再帰的に行なわれる。例えば次に示す連結が存在する場合、終端オブジェクトである Holder D を消去するだけで全てのオブジェクトおよび関連するデータ領域がメモリから解放される。

Source A → Buffer B →
Filter C → Holder D

ただしオブジェクトのリンクは一般には直列とは限らないので、自動的な消去に関してリファレンスカウンタを利用する。各オブジェクトのデストラクタはリファレンスカウンタを減算していき、カウンタが 0 になった時点で実際の消去を実施する。順次処理の性質上循環構造を成すリンクはあり得ないため、リファレンスカウンタによる消去処理は問題なく動作する[2]。

次に、画像フレームの更新処理における同期の確保に関したリファレンスカウンタの利用について説明する。

画像処理はリンク構造の最後尾のオブジェクトに対して `malib_holder_increment_frame()` を呼び出すことで開始される。同関数では前段の処理オブジェクトへ参照を辿り再帰的に処理関数を呼ぶ。リンク構造の先頭、すなわちソースオブジェクトまで関数呼び出しが至った時点で、そのソースでは新規フレームを生成する処理が行なわれる。そして、呼び出しを戻りつつ新たに生成されたフレームのデータに各フィルタ処理を加えることにより、前節で示したとおりの一連の画像処理が行なわれる。

このように、動画像処理は各時刻において逐一画像フレームを更新していくことで進行する。その際、正確な解析を実現するには各フィルタの処理対象が異なっていてはならない。比較処理を行なう場合、比較対象が異なっていては意味がないので、この条件は顕著な問題となる。

先に示した Source A から Holder D までの一連の処理に類した直列処理の場合は、端点から端

[2]意味のないデータ構造として、循環構造をなす参照構造を構成することは実際には可能である。ただしこのようなオブジェクトは MAlib からは消去不可能になるので注意が必要である。

点までの参照関係は全て一対一の対応となっており、問題はない。ところが、図 4 に示すような複雑な構造の場合、終端の Holder K から始点の Source A までのパスは複数存在する。リファレンスカウンティングなしの単純な関数呼び出しで処理を行なうと、次に示す呼び出し順序で処理が行なわれることになる。

K → J
→ D → C → B → A
→ H → G → F → E → B → A
→ I → E → B → A

ここでオブジェクトに付加されたアンダーラインは、一回のフレーム更新処理に関して複数回呼び出されていることを示す。上記の例では、Holder K に対する一回の更新処理について Source A が 3 回呼ばれているため、Buffer D からの系列、Buffer H からの系列、Buffer I からの系列の全てが異なるフレームデータに基づいた処理を行なうことになり問題である。

そこで、このような複雑なリンク構造においても、元データの同一性の確保すなわち処理の同期を保持するために、通常のリファレンスカウンタに加え一時的に値が増減するテンポラリ・リファレンスカウンタを導入した。さらに次に述べる規則の導入により問題解決を図ることができる。なおこれらのカウンタは `MalibObject` において、メンバ変数 `refcount` および `refvar` として実装されている。

テンポラリカウンタ `refvar` の取扱いに関する規則は以下のとおりである。

1. `refvar` の初期値は `refcount` に等しい。すなわち通常のリファレンスカウンティングを行なう場合は、`refcount` の増減に合わせて `refvar` の値の増減も行なう。

2. 終端のオブジェクトからリンク構造を辿りつつ画像処理を進める際には、`refvar` と `refcount` の値が等しい場合のみ、前段へ処理を進める。始点オブジェクトの場合は、`refvar` と `refcount` の値が等しい場合のみ新規の画像を生成する。`refvar` と `refcount` の値が等しくない場合は、キャッシュされている画像を利用し後段へ処理を進める。

3. 一回の呼び出しにつき、`refcount` の値をひとつ減算する。`refcount` の値が 0 になった時点で `refcount` の値に初期化する。

上記の規則を適用すると、先ほどの呼び出し順序は次のように修正される。

K → J
→ D → C → B → A
→ H → G → F → E → (B)
→ I → (E)

括弧で示した箇所は、前段に対する処理の呼び出しを実施せずに既にあるデータに対して処理を行なっているものである。この呼び出し順序によると、Source A の新規画像作成処理は一度しか行なわれない。また例えば Filter E におけるフィルタ処理の適用画像は、既に Filter C から Buffer B の関数が呼び出された結果として生成され、Buffer B にキャッシュされているものと同じデータである。このように全てに対して同じソースからのデータが、フレームがずれることなく順次全ての処理オブジェクトに対して渡されるようになっていることがわかる。

7 アプリケーション例

MAlib を利用したアプリケーションの例題として、道路を通行する車両の認識を行なうアプリケーションの構築と実験を行なった。アプリケーション `objtraq` は、グレースケールの入力動画像から移動物体を抽出する。

`objtraq` に入力する動画像のスナップショットを図 6 に示す。

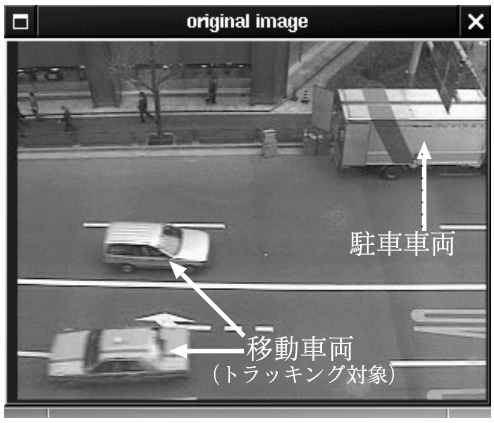

図 6: オリジナル画像

この瞬間は、画面右上に駐車しているトラックと、画面中央を左から右へ通行している小型車、および画面下側を右から左へ通行しているタクシーの 3 台の車両が画像に表示されている。`objtraq` は移動物体のみを認識するため、中央左の小型車と左下のタクシーが認識の対象となる。

`objtraq` を実行すると、上記の入力画像の他、

認識途中のパターンを入力画像に重ねて表示するマスク画像 (図 7) と認識結果を表示するトラッキング画像 (図 8) の合計 3 枚のウィンドウが表示される。

図 7: マスク画像

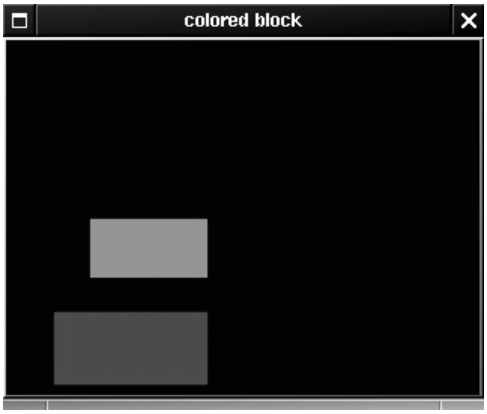

図 8: トラッキング画像

図 8 には、移動している 2 台の車両が認識されていることがわかる。実際の表示においては、リアルタイム認識として連続的に認識処理が行なわれた結果が示される。

上記の処理を実現するコードの主要部を図 9 に抜粋する。ライブラリ MAlib が高度にモジュール化されているため、MAlib の各機能を接続することで実現する画像処理の主要部のコード量は約 70 行と、非常にコンパクトな記述となっていることに注目されたい。

また objetraq の認識処理を構成する画像処理オブジェクトのリンク構造を図 10 に示す。画像処理のプロセスが、Source → Filter → Buffer → … → Holder(表示) といったリンク構造として示されている。

なお認識処理の中心は図 10 における filter1、filter2、filter3、および filter5 の各フィルタであり、順に空間平滑化、フレーム間差分による移動物の検出、ブロックセグメンテーション、移動物体のトラッキングの各処理を実現するものである。

```
{
  MalibSource *filter1, *filter2, *filter3, *filter4, *filter5;
  MalibBuffer *buf0,    *buf1,    *buf2,    *buf3;

  /* first buffer is ring buffer */
  buf0    = (MalibBuffer*) malib_ringbuf_new_with_source (6, src);

  {
    /* averaging filter for noise suppression */
    static int coef[3][3] = { {  1, 1, 1 }, { 1, 1, 1 }, {  1, 1, 1 } };
    filter1 = (MalibSource*) malib_spatial3x3_new_with_coef (buf0,coef);
    buf1    = (MalibBuffer*) malib_ringbuf_new_with_source (6,filter1);
  }

  {
    /* moving object extraction using frame differential */
    filter2 = (MalibSource*) malib_greydiff3_new_with_threshold (buf1,
                arg_threshold);
    buf2    = (MalibBuffer*) malib_plainbuf_new_with_source (filter2);
  }

  {
    /* block segmentation filter for BW image */
    filter3 = (MalibSource*) malib_genericfilter_new_with_config
        (buf2, MALIB_FRAME_COLORMODEL_BW, NULL, blockseg);
    buf3    = (MalibBuffer*) malib_ringbuf_new_with_source (3,filter3);
  }

  {
    /* masking captured image (buf0) with extracted regions (buf4) */
    filter4 = (MalibSource*) malib_masking_new_with_bufs (buf0, buf3);
  }

  {
    /* small BW frame */
    MalibFrame* frame3 = malib_buffer_get_current_frame (buf3);
    MalibFrame* frame5 = malib_frame_new
        (MALIB_FRAME_COLORMODEL_RGB,
         frame3->width  / get_param("BLK"),
         frame3->height / get_param("BLK"),
         8, NULL);

    /* moving object tracking filter */
    filter5 = (MalibSource*) malib_genericfilter_new_with_config
        (buf3, MALIB_FRAME_COLORMODEL_BW, frame5, track_objects);
  }

  {
    MalibGtkDisplay* darray[4];

    /* create GTK windows */
    darray[0] = malib_gtkdisplay_new_with_source (src);
    darray[1] = malib_gtkdisplay_new_with_source (filter5);
    darray[2] = malib_gtkdisplay_new_with_source (filter4);
    darray[3] = NULL;

    /* set window titles */
    gtk_window_set_title (malib_gtkdisplay_get_window (darray[0]),
                          "original image");
    gtk_widget_set_usize (GTK_WIDGET(malib_gtkdisplay_get_window
                          (darray[1])), arg_width, 3*arg_width/4);
    gtk_window_set_title (malib_gtkdisplay_get_window (darray[1]),
                          "colored block");
    gtk_window_set_title (malib_gtkdisplay_get_window (darray[2]),
                          "masking");

    /* starting image processing and display results */
    malib_gtkdisplay_auto_play2 (darray);
  }
}
```

図 9: objtraq の認識処理部

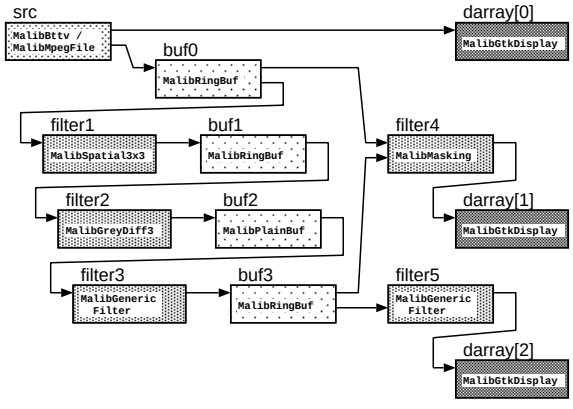

図 10: objtraq における処理オブジェクトの構造

8　今後の計画

最後にプロジェクトの今後の計画について説明する。プロジェクトの将来計画を説明するにあたり、本プロジェクトの背景とビジネスモデル、企業活動におけるオープンソース開発のあり方について補足する。また今後解決すべき技術的な課題を挙げる。

本プロジェクトは情報処理振興事業協会による平成 12 年度未踏ソフトウェア創造事業の一環として実施された。未踏ソフトウェア創造事業は当協会その他による一般的な助成事業と異なり、その支援対象が組織に対してではなく個人に対するものであるという重要な特長を有している。

企業の業務においては、その開発成果物の著作権は顧客あるいは所属組織に属する場合が一般的である。しかし本件に関しては当該事業の目的に応じ、知的所有権については開発者個人に属するものとしている。したがって、オープンソースソフトウェアとして公開する点に関する業務上の制約は存在しなかった。

一方、現状の様々な事例に散見されるように、オープンソースソフトウェアに基づいたビジネス展開については、多くの事業者がビジネスモデルを模索中の段階である。筆者の所属する組織でも、オープンソース活動を支援することで利益を上げる様々な方法の検討が行なわれている。

そのなかで筆者らは、オープンソースプロジェクトとして本プロジェクトの開発を進めるとともに、本プロジェクトの成果物である MAlib をベースとした個別のアプリケーション開発あるいは MAlib を利用したサービスの提供で収益を得る事業構造を想定している。なおライブラリ利用者の便宜を図るためといった理由に加え、このような柔軟な運用にも合致させるためにも、本プロジェクトにおいてはライセンス形態として LGPL を採用する必要があった。

MAlib はまだ基本的な枠組みができあがったばかりの段階であり、まだ実装が不十分な箇所もある。重要な検討項目として、音声と画像を同期させたデータ構造の定義、様々なパラメータや処理結果データを伝搬する方法の検討、画像データの差異 (サイズや色深度など) を吸収し統一的に扱う手法の確立などが解決すべき項目として残されており、今後の課題である。

9　まとめ

本プロジェクトでは汎用動画像処理ソフトウェアライブラリ MAlib を、C 言語を用いたオブジェクト指向設計によって汎用性と保守性を両立させつつ実現した。またその利用例として、移動物体認識アプリケーションが簡単に実装できることを示した。さらに本プロジェクトの位置づけを示し将来展望を説明するにあたり、補足として企業におけるオープンソース開発について述べた。

MAlib は現在、次の URL で公開している。

`http://www.malib.net/`

先に述べたとおり本プロジェクトは引き続き開発を行なっており、共同開発者やユーザも鋭意募集中である。興味のある方はぜひとも上記 URL を参照していただき、コメントを頂ければ幸いである。

参考文献

[1] 田村, 坂根, 富田, 横矢, 金子, 坂上. “ポータブル画像処理ソフトウェア・パッケージ SPIDER の開発”, 情報処理学会論文誌, 23 巻 3 号, pp.321-328, 1982-5.

[2] H. Tamura, S. Sakane, F. Tomita, N. Yokoya, M. Kaneko, and K. Sakaue. “Design and implementation of SPIDER - A transportable image processing software”, Computer Vision, Graphics, and Image Processing, Vol.23, No.3, pp.273-294 (1983)

[3] Davis, J. and Bradski, G. “Real-time Motion Template Gradients using Intel OpenCV”, IEEE ICCV’99 Frame-Rate Workshop, 1999.

[4] The Vision Technorogy Group (Microsoft Research). “The Microsoft Vision SDK”, `http://research.microsoft.com/projects/VisSDK/`, May 2000, Viewed May 18, 2001

[5] Jhon Cristy. “ImageMagick - Image Comversion, Editing, and Composition”, `http://www.imagemagick.org/`, Viewed May 18, 2001

[6] Bryan Henderson. “Netpbm home page”, `http://netpbm.sourceforge.net/`, Apr 2000, Viewed May 18, 2001

[7] 飯尾淳. “Linux による画像処理プログラミング”, オーム社, 2000.

[8] Tony Gale and Ian Main. “GTK+ 1.2 Tutorial”, `http://www.gtk.org/tutorial/`, Mar 2001, Viewed May 18, 2001