平成 24 年度　特別研究報告書

複数センサを用いた画像領域分割

龍谷大学　理工学部　情報メディア学科

T090448　広瀬　大樹

指導教員　三好　力　教授

内容梗概

画像領域分割とは，画像を構成する物体を検出して，それらの構成要素を領域として捉えるために，一様な画素特徴を持つ小領域に分割する処理のことである．物体の領域抽出が行えるようになると，画像認識・画像合成等様々な技術への応用が期待できる．人間は物体を 3 次元的に捉えることができるためこの処理を自然に行うことが出来るが，機械は平面的に画像を捉えるため画像が多少幾何学的変換をなされただけでもこの処理を行うことが非常に難しくなってしまう．

従来の手法では画像の RGB 情報を用いて領域分割を行うため，画素的に似た領域の分割が非常に難しい．また，リアルタイム処理を行う場合，色情報だけでなく，距離情報も同時に用いて領域分割を行い RGB 画像処理の負担を減らすことで，より柔軟な画像処理が期待できる．

本研究では，複数センサの搭載されている Kinect を用いた距離センサとカメラ画像による物体領域分割手法を作成した．

目次

第 1 章 はじめに

1.1 研究背景

人間は，ある対象物を見たとき，その 3 次元形状，サイズ，テクスチャ等の情報を瞬時に分析し，その対象物が，分類としては何に属するか，その分類の中ではどんな特徴があるのか，何かほかの分類に似ているか，何に使えそうか等の複数のカテゴリーの認識を同時平行的に行うことが出来る．しかし機械は，画像を多少幾何学的変換が行われただけでも物体抽出・認識を行うことが非常に難しくなる．これは，人間が物体を 3 次元的に捉え，理解するのに対し，機械は平面的に画像を捉えることに起因すると考えれば，機械で人間と類似の高度な認識を行うには処理に人間と類似のアプローチを行う必要がある．

画像処理分野において，画像の領域分割を行うことは画像に写っている情報の理解を行うために非常に重要である．また，領域分割を行えるようになることで動的コンテンツの制作も容易になると考えられる．さらに，画像の領域分割は，画像認識・画像合成等様々な技術への応用ができることや，特定の人物の領域を追尾することによる警備への利用，掃除ロボットの物体認識による行動の多様化など，社会での新しいサービスでの利用が考えられる．掃除ロボットなどリアルタイム処理が求められる場面では RGB カメラのみでなく，複数センサを用いて物体認識や場面認識を行うことで，多用な行動決定の材料にすることが期待できる．

本研究は赤外線距離センサと RGB カメラを用いて 3 次元形状，テクスチャの 2 つの特徴から，複数のカテゴリの認識が可能なシステムの構築を目指し，その基礎となる物体認識処理について上記の特徴の有効利用を試みる．

1.2 物体認識

物体認識とは画像をシステムに入力し，入力に対し適切なラベルを付与する過程のことをいう．一般に実世界画像に対する物体認識は大きく分けて同定 (identification)と分類 (classification)の 2 種類の認識がある．同定とは画像内の物体を個々に区別する認識であり，入力画像とデータベース内のモデルと照合を行い，画像内の物体がデータベース内に存在するか，またどのモデルと一致するかを出力結果とする．一方分類は画像内の物体の種類を区別する認識であり，人間が定めたクラスと画像内の物体を対応付け，対応したクラス名を出力結果とする．物体認識で行われている研究は特定物体認識と一般物体認識の 2 種類に分類することができる．特定物体認識は同定，一般物体認識は分類の分野を扱っている．物体認識処理の流れは学習時と識別時のどちらでもデータ入力，特徴抽出，識別記の順に処理されていくのが一般的である．

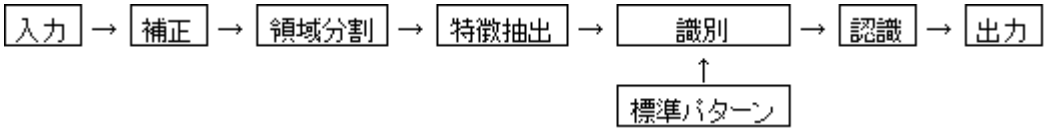

図 1.1. 物体認識システム

1.3 領域分割

画像の解析，理解を行う場合に画像中に含まれる対象を分離，抽出することが必要となる．この処理を領域分割を呼ぶ．エッジや線の抽出，追跡して領域を分割する方法も考えられるが，同一領域は類似の特徴を持つという性質を利用して，領域を分割する手法が一般に用いられている．後者の手法では対象の境界線は必ず閉じたものとして得られ，ノイズの影響を受けにくいという利点がある．

ここでいう領域とは画像中で共通の性質を持つ部分のことをいう．通常，画像中でひとかたまりになっているが，お互いに離れた複数の部分から構成される場合もある．[1]

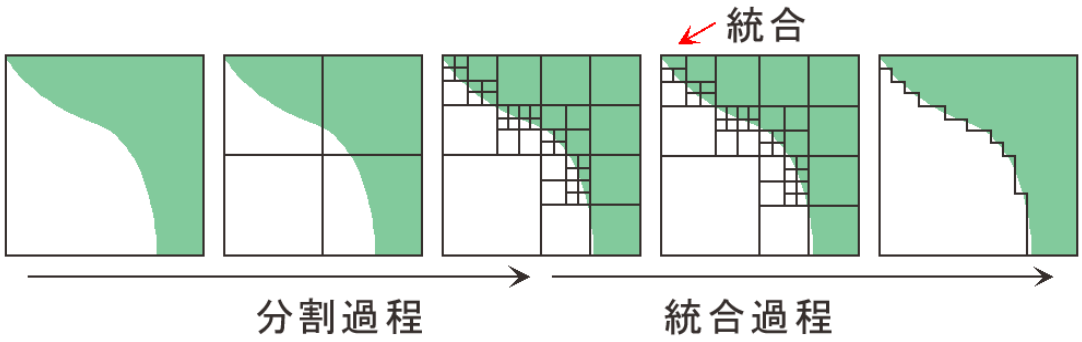

図 1.2 領域分割概念図(分割統合法)

1.4 研究目的

物体認識を行うには，画像内の物体領域の抽出，分割が非常に重要である．従来の領域分割手法には RGB 画像を用いて画素に基づくクラスター分析による画像分割や画像の局所的な類似性に基づいた分割手法，領域間のエッジに基づいた分割手法などがある．RGB 画像で物体の領域分割を行う場合，画像内に物体が複雑に配置されていると光の当たり具合や隣接物体との色の類似度によって境界線を判断することが非常に難しいため，画像内の物体認識を行うことができない．距離センサ画像に基づく領域分割は光や色に左右されないため領域抽出が容易に行え，複雑な画像の物体認識が行うことができるであろう．

本研究では Kinect による距離画像を用いたフィルタを作成し，RGB 画像とあわせた画像領域分割システムを作ることを目的とする．

第 2 章 既存手法

2.1 クラスタリング手法

2.1.1 K-means 法

K-means 法は非階層型クラスタリングでよく用いられるクラスタリング手法で，あらかじめ指定したクラスタ数にデータを分割し，そのクラスタ内部で中心をとり再度クラスタに分割しなおすという方法を繰り返す．

K-means 法のアルゴリズムを以下に示す．
①K 個のクラスターの中心をランダムに設定する．
② それぞれの個体を最も近い中心に割り当てる．
③ クラスターごとに中心を計算しなおす．
すべてのクラスター中心が変化しなければ終了．
それ以外は Step 2 へ戻る．[2]

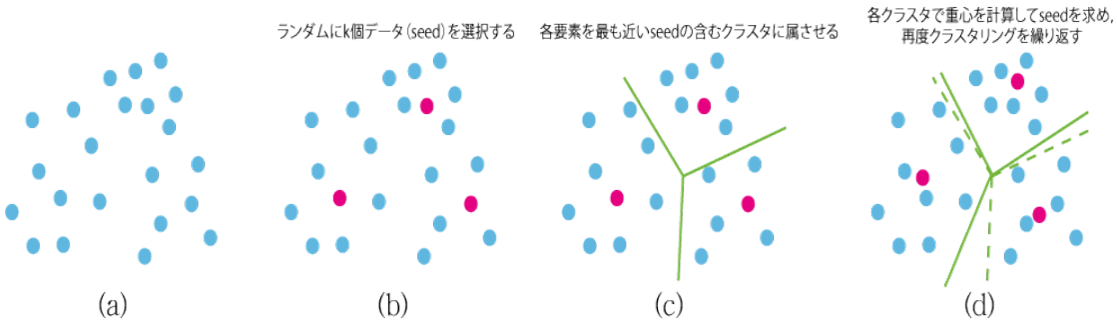

図2.1 非階層型クラスタリング

図2.2 k-means法による減色処理(元画像,k=5,k=10)[8]

2.1.2 平均値シフト法(Mean-shift)

関数 f(x)の初期値周辺のある区間の傾きにより，f(x)の値が大きくなる方向へ区間中心の移動を繰り返すことで，初期値周辺において f(x)が極大になる点を求める手法.

極大値の位置を決定すれば，それに対応するクラスタは特徴空間の局所的な構造に基づいて描きだされる. 分布密度の極大値は潜在的な密度関数の勾配が 0 である点に位置する. 平均値シフトアルゴリズムは密度を推定することなく零点の位置を見つけ出し最頻値探索とクラスタリングに対して有効な手法であると考えられる. [3]

平均値シフト法のアルゴリズムを以下に示す.

①画像を色情報，輝度情報等でいくつかの部分に分類する.

②データから仮に探索ウィンドウの初期位置を決定する.

③各々の初期位置から探索ウィンドウの平均位置を計算する.

④同じ状態やピークを持つウィンドウを合併する.

⑤合併することにより横断されたデータが同じクラスタに属する.

図 2.3 平均値シフト法による画像領域分割

2.2 輪郭抽出

エッジ抽出[4]

1.Sobel フィルタ

空間 1 次微分を計算して，輪郭を抽出するフィルタ. そのため，1 次微分に平滑化作用を持つ. アナログ画像の場合，ある一定画素位置を右から微分しても，左から微分しても同じ値であるが，離散データであるデジタル画像の場合，右の画素から見るか，左の画素から見るかでは値が異なってしまう. そこで，sobel フィルタは表 2.1,表 2.2 のように中心画素に対して縦横に重み付けを行うことによってエッジを抽出している.

2. ラプラシアンフィルタ

空間 2 次微分を計算して，輪郭を検出するフィルタ．このフィルタは輝度の差分の変化量が大きくなっている部分を抽出するフィルタ．一般的にエッジ検出や先鋭化に用いられる 2 次微分フィルタの一種．エッジの重み付けには表 2.3,表 2.4 のように重み付けを行う．

3. canny 法

x 方向と y 方向にガウス関数をたたみこむ．次に注目がその周囲 8 点から勾配の最大位置を検出．最後に勾配の大きさに合わせて閾値処理をして，それをエッジ検出結果とする．弱いエッジも検出しやすいため未検出や誤検出が少ないのが強み．

表 2.1 sobel 縦方向フィルタ

-1	-2	-1
0	0	0
1	2	1

表 2.2 sobel 横方向フィルタ

-1	0	1
-2	0	2
-1	0	1

表 2.3 4 方向の場合

0	1	0
1	-4	1
0	1	0

表 2.4 8 方向の場合

1	1	1
1	-8	1
1	1	1

2.3 顕著性マップ(Sailency Map)

画像中の視覚的注意を引く領域を抽出する手法であり，一般的に画像中の物体は背景に比べて顕著性が高いと考えられる．人間の視覚情報処理の初期段階では，いくつかの単純な視覚的特徴が処理され，複数の特徴マップ(feature maps)として表現される．その後，それらの特徴マップが統合され，顕著性マップ(Saliency Map)として表現されるという特徴統合理論に基づき Saliency Map はモデル化されている．

顕著性マップは被験者を必要とせず，画像の物理的な特徴を解析するだけで注意が向けられる位置を推定することができる．図 2.6 は，顕著性マップの算出過程に関する概念図を示す．顕著性マップの算出過程において，特徴マップ生成と特徴マップ合成がある．特徴マップ生成では，画像データに対して6つの視覚属性それぞれに関する画像解析を行う．視覚属性には，色，明度，方位，コントラスト，点滅，運動がある．さらに，特徴マップ合成では，視覚属性ごとに生成した特徴マップを線形和し，顕著性マップを算出する．このように顕著性マップは画像の物理的な特徴からの注意の向けられやすい領域を求めることができるが，実際にその画像を人間に見せたとき，顕著性の高い領域に人間の注意が向けられるとは限らない．

また，顕著性マップの算出過程において，各特徴マップを線形和にて合成する際，各視覚属性に重みパラメータを付与することができる．すなわち，同じ動画像データであっても，これらの重みパラメータを変化させることで最終的な顕著性が変わることになる．注意の特性に応じた最適な重みの配分があると考えられるが，これらについては十分な検討がなされていない．[5]

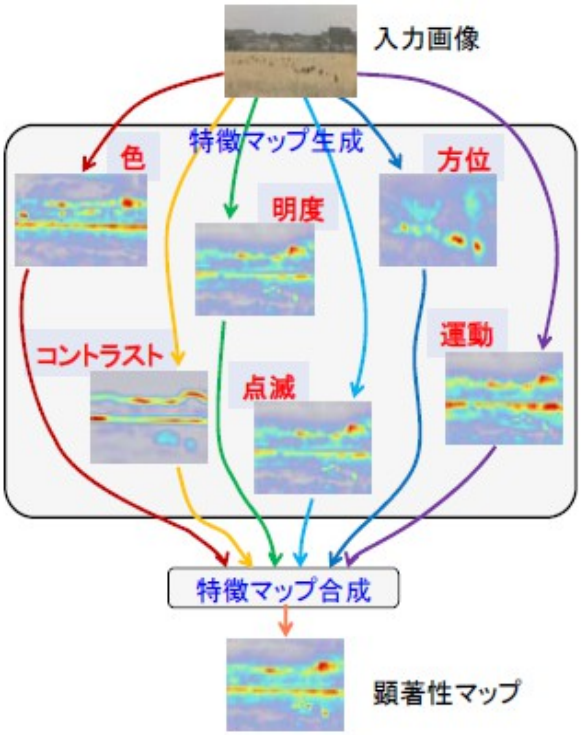

図 2.4 顕著性マップの概念図

第 3 章 Kinect とは

Kinect は Microsoft から販売されている Xbox360 専用のゲームデバイスである．2010 年 11 月 4 日に米国で発売され，同年 11 月 20 日に日本でも発売された．Kinect はコントローラを使わずに操作ができる体感型のゲームシステムが特徴で，ジェスチャーや音声認識を用いた直感的なプレイが可能である．このようなユーザインタフェースは Natural User Interface(NUI)と呼ばれ，従来のデバイスは，ゲームを動かすために情報を入力するコントローラを使っていたのに対し，Kinect は内蔵されている 3 つのセンサを用いて高速に演算するプロセッサによって処理することで，人間の検出，プレイヤーの姿勢の認識，リアルタイムの動きの把握をし，ゲームの操作を可能としている．[6]内蔵されているセンサには，映像の出力や写真を撮るためにカラー画像を出力する RGB カメラ，Kinect から対象までの距離を測定できる近赤外線距離画像センサ，音声を認識するための 4 つのマイクセンサが搭載されている．近赤外線距離画像センサは近赤外光パターンを広範囲にレーザを照射する近赤外光プロジェクタとレーザ照射された近赤外光パターンを撮影し，撮影された画像から奥行きを計算する近赤外カメラ(3D 奥行きカメラ)で構成されている．

近赤外線距離センサでは三角測量を利用した計測法を採用している．まず，Kinect に搭載された近赤外レーザから対象物にスポット光をランダムに配列したようなパターンを複数投影し，対象物に映ったパターンをカメラでとらえる(図 3.1)．そして，とらえたパターンのカメラからの見込み核を求め，レーザから対象物にパターンを照射した角度と，レーザとカメラの距離を基に，対象物までの距離を算出する．パターン内のスポット光を区別できれば，1 回の投影によってカメラがとらえた画面内の物体の複数店の距離計測が可能となる．[7]

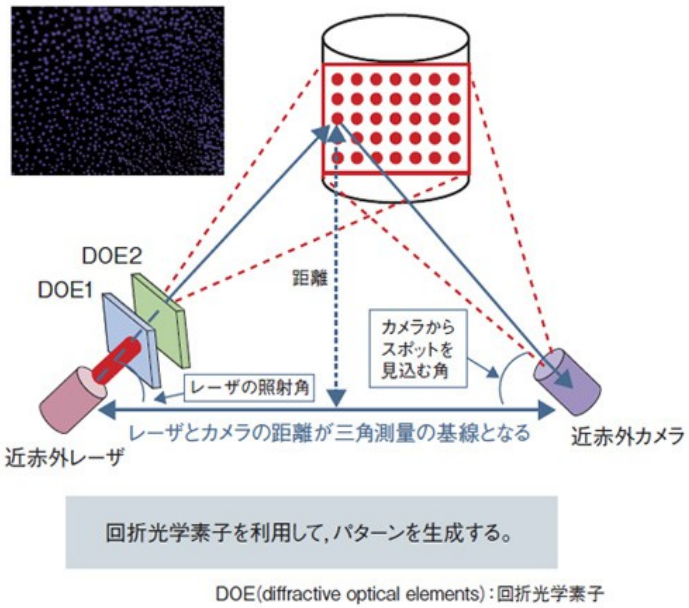

図 3.1 三角測量による対象物の距離測定

また，Kinect にはUSB インタフェースが付属されていることため，コンピュータにも接続可能であり，開発元のMicrosoft は，非商用の場合に限ってコンピュータでもKinect を利用することを認めており，「内部アルゴリズムにアクセスしないこと」と「Xbox 向けのゲームの不正利用に使用しないこと」を条件として開発を許可している．さらに，発売当初はMicrosoft から公式なデバイスドライバの提供予定はなかったが，2011 年6 月16 日に windows7 での使用を可能にするソフトウェア開発キット「Kinect for Windows SDK(MS SDK)」のベータ版をリリースした．また，Kinect をMicrosoft と共同開発したPrimeSense などが中心となって実装しているオープンソースライブラリの「OpenNI」によってKinect を含めたNI デバイス全般を扱うことができる．ライセンスは GPL およびLGPL である．開発環境としては，NI デバイスのドライバやインターフェースを担当するセンサ・モジュール，およびユーザ/スケルトントラッキングや各種ジェスチャーを認識できる「NITE」をOpenNI と併せてインストールするのが一般的である．MS SDK とOpenNI の仕様を表3.1 に示す．

表3.1 MS SDK,OpenNI の仕様[9]

項目	公式SDK	OpenNI
対応OS	Windows 7(x86,x64) Windows 8 Developper Preview	Windows XP, Vista, 7(それぞれx86,x64) Windows 8 Developper Preview Linux(Ubuntu)(x86,x64) Mac OS Android
開発言語	C++,C#	C,C++,C#,Java
対応デバイス	Kinect	Xtion Pro,Xtion Pro LIVE(公式),Kinect(非公式)
カメラ解像度	1280x1024,640x480	1280x1024,640x480
距離解像度	640x480	640x480
距離範囲	850mm-4,000mm	500mm-10,000mm
ユーザー認識解像度	320x240,80x60	640x480
スケルトントラッキングのためのポーズ	不要	OpenNI 1.4.0.2以降では不要
ユーザートラッキング数	7人	仕様なし
スケルトントラッキング数	2人	仕様なし
部分的なトラッキング	不可	可
ミラー機能	不可	可
カメラと距離のズレ補正機能	可	可
カメラ、距離の保存、再生	不可	可
ジェスチャー検出機能	不可	可
複数デバイスの操作	可	可

OpenNI には OS が Windows に限定されない，画像をミラー反転させたり，RGB 画像と深度画像のビューポイントを一致させる関数や，Kinect から取得したデータをファイルに書き出せるなどの利点がある．また，現段階では用いていないが今後の研究で部分トラッキングやカメラ画像，距離データの保存，再生を使う必要がある．また，移動ロボットへの実装も考えているため本研究では OpenNI を使って研究を進めた．

図 3.2 Kinect 外観

第 4 章 提案手法

3D 特徴を用いた画像領域分割

4.1 概要

画像領域分割では，RGB 画像の色情報，輝度情報などを使った領域分割が行われているが画面の全領域を RGB データのみで分割すると物体の配置や光の反射具合によって非常に難しくなる場合がある．本研究では，ゲームコントローラ Kinect に搭載されている近赤外線距離画像センサを用いてフィルタを作成し画像の切り出しを行うことによってより複雑に物体が配置された画像から中心に配置された物体の領域を容易に抽出できるシステムを提案する．画像の切り出しのためのフィルタを作成するために，まず，Kinect の近赤外線距離画像センサから得られる距離センサ情報を用いて画面内の有用な特徴量を決定する．次に，決定した特徴量を用いてフィルタを作成し，カメラ画像から RGB 画像の切り出しを行うシステムを開発する．

4.2 画像フィルタの作成

画像領域分割を行うために Kinect から得た情報を用いて，領域分割に用いる特徴量を決定する必要がある．特徴量の決定には Kinect の近赤外線距離画像センサから得られる深度画像を利用する．深度画像から距離データを用いて有効な情報を決定し深度画像を 2 値化する．この 2 値化画像をフィルタとして画像領域分割を行う．　特定の範囲内に物体が存在する座標を 1，存在しない座標を 0 とし，2 値化画像の作成を行う．

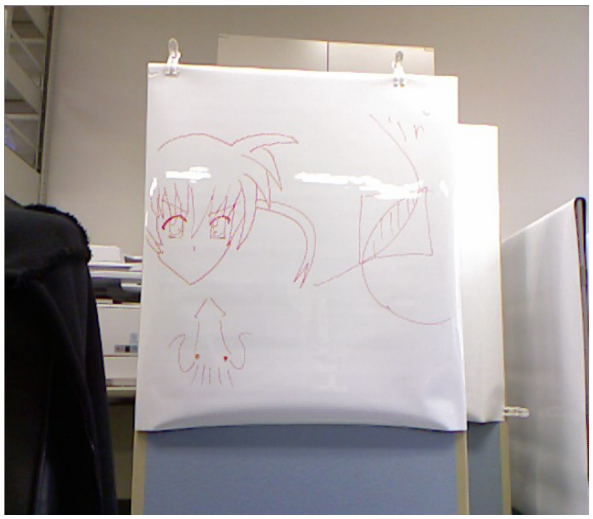

図 4.1 取得したカメラ画像

図 4.2 深度センサから作成したフィルタ画像

4.3 フィルタを用いた画像領域分割

画像領域分割には 4.2 節で作成した 2 値化画像をフィルタとし 1 となった部分とカメラ画像を重ねる．重なった部分を切り出す．距離画像を元に切り出しを行っているため，RGB 画像では領域分割が難しい画像でも容易に領域分割が行える．

図 4.3 マスク処理によって切り出した画像

第 5 章　実験と評価

フィルタの作成をするために，プログラムを作成し，近赤外線距離画像センサの深度画像から距離データを取得した．取得した距離データを基に深度画像を 0 と 1 で 2 値化し，2 値化した画像をマスクとしてカメラ画像から画像の切り出しを行った．領域分割のために用いた距離画像は近赤外線距離画像センサによって作成された画像であるため，光の具合などに左右されない距離情報が取得でき，物体領域の抽出が容易であることが期待できる．

提案手法の有効性を検証するため，RGB 画像を基に作成したフィルタを用いて目標物体の抽出を行う従来手法と，提案手法の比較を行う．

本実験は目標物体のみが画面内に配置された場合と，目標物体とは別の物体が画面内に複数存在する場合の 2 通りの実験を従来手法，提案手法両方で行った．処理の概要を 5.1，5.2 に示す．

5.1 従来手法の実験方法

従来手法で領域分割が行えているかを確認するために，以下の手順で実験を行った．

- Kinect から約 1.5m 離れた位置でカメラ画像の中心となるように目標物体を配置．
- RGB カメラから RGB 画像を取得．
- RGB 画像をグレースケールに変換．
- 画像の中心座標の色を基に画像を 2 値化しフィルタを作成．
- フィルタを用いてカメラ画像から目標物体の切り出し．
- 切り出された画像を出力．

5.2 提案手法の実験方法

提案手法で領域分割が行えているかを確認するために，以下の実験を行った．提案手法のフローチャートを図 5.1 に示す．

- 従来手法の実験(5.1 節)と同様に物体を配置．
- RGB カメラから画像を取得．
- 近赤外線距離センサから距離情報を取得．
- 壁，床の距離情報を背景として背景差分を行う．
- 特定の距離範囲内を 1，その他領域を 0 で 2 値化しフィルタを作成．本実験では有効距離を 1.0m～1.5m とした．2 値化画像は 1 が白，0 が黒で出力される．
- 距離画像はカメラ画像と座標にズレがあるため，OpenNI により座標補正を行う．

- フィルタを用いてカメラ画像から目標物体の切り出し.
- 切り出された画像を出力.

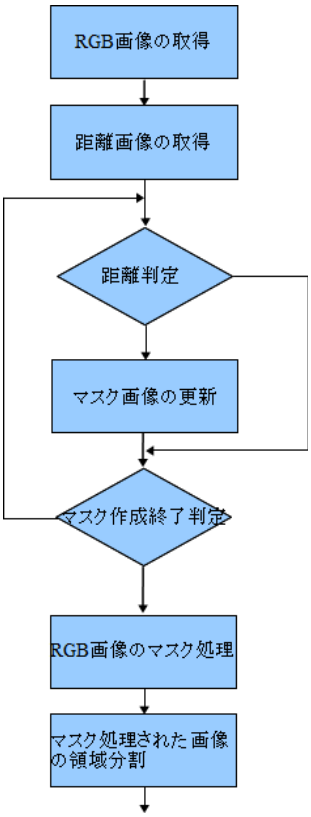

図 5.1 提案手法のフロー

5.3 実験環境

本実験では目標物体の認識距離を 1.0m～1.5m として認識を行った. Kinect は地面より 0.7m の高さに配置, 全物体は Kinect の認識可能距離である 0.5m~5.0m の位置に配置した. 開発環境は Visual C++ 2010, RGB カメラからのカメラ画像の取得, 近赤外線距離画像センサからの深度画像の取得に OpenNI 1.5.2.23, 取得した画像の変換処理, 表示については OpenCV 2.2.0 を用いて行った. [10]

5.4 実験結果と評価

RGB カメラ画像を基にフィルタを作成し RGB カメラ画像を切り出した場合と，距離センサを用いてフィルタを作成し RGB カメラ画像を切り出した場合の出力画像を比較し，どちらが目標物体の抽出が正確に行えているかを評価した．また，画像取得から切り出された画像の出力までの処理時間の比較し，どちらの処理が早く，リアルタイム処理に適しているかの評価も行った．

5.4.1 目標物体のみが画面内に配置された場合の画像分割結果

実験に用いた元画像を図 5.2，従来手法の出力結果を図 5.3，提案手法の出力結果を図 5.4 に示す．従来手法では目標物体が光を反射してやや白くなっている部分の切り出しがうまくいっていないが，提案手法では目標物体全体の切り出しが正確に行われている．また，従来手法の出力結果は壁の一部が切り出されてしまっているが，提案手法では目標物体のみの切り出しができている．以上から，目標物体のみ配置した場合，提案手法の有効性が確認できる．

図 5.2 画面内に目標物体のみの場合の元イメージ

図 5.3 画面内に目標物体のみの場合の従来手法のフィルタ画像(左)と出力画像(右)

図 5.4 画面内に目標物体のみの場合の提案手法のフィルタ画像(左)と出力画像(右)

5.4.2 画面内に物体が複数配置された場合の画像分割結果

実験に用いた元画像を図 5.5，従来手法の出力結果を図 5.6，提案手法の出力結果を図 5.7 に示す．従来手法では 5.4.1 の結果と同様に目標物体の全体がうまく切り出されず，さらに目標物体以外の物体も切り出してしまっている．提案手法では，目標物体の全体が正確に切り出され，他の物体の切り出しも行われていないため，提案手法は有効であると言える．

図 5.5 画面内に物体を複数配置した場合の元イメージ

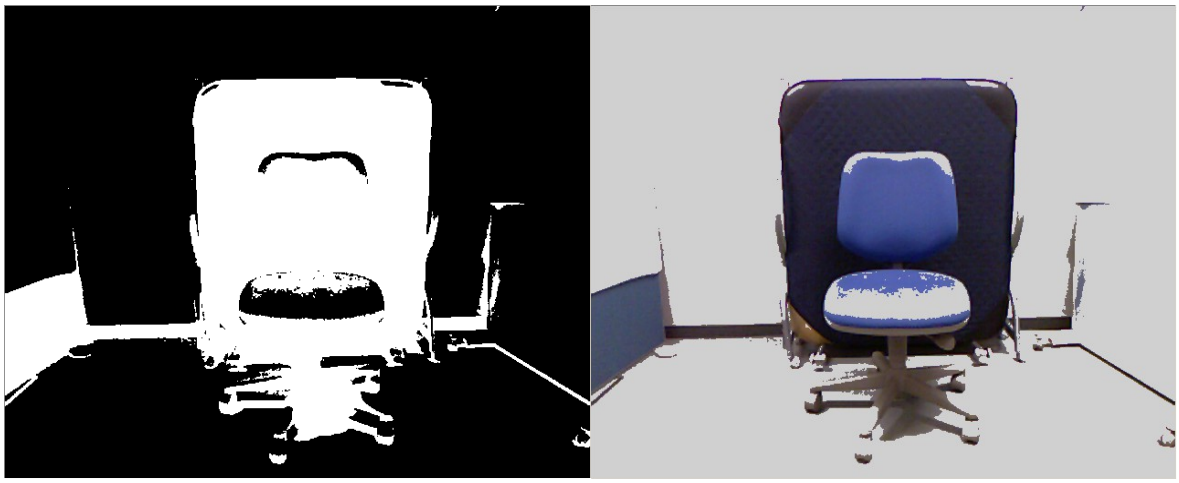

図 5.6 画面内に物体を複数配置した場合の従来手法のフィルタ画像(左)と出力画像(右)

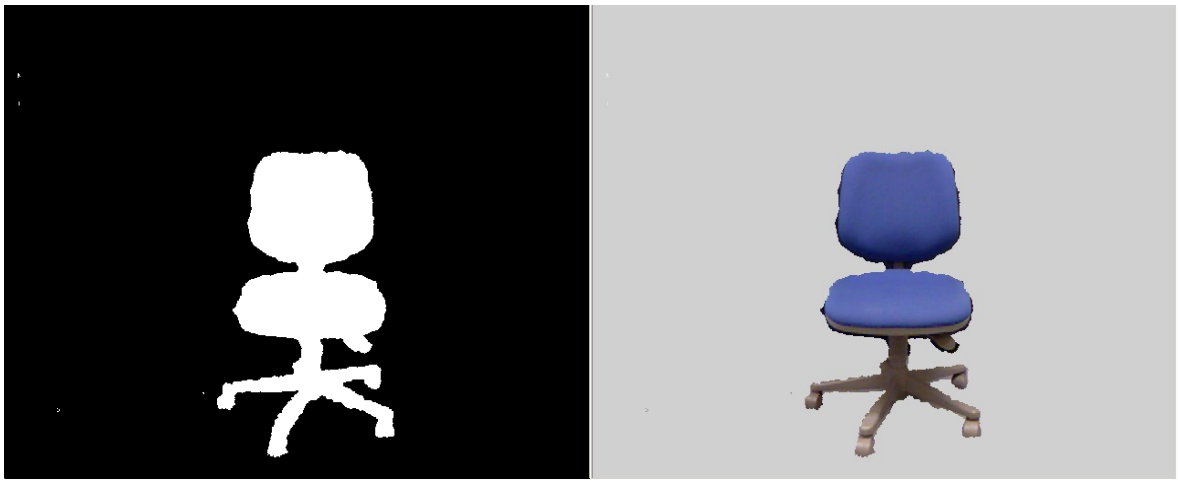

図 5.7 画面内に物体を複数配置した場合の提案手法のフィルタ画像(左)と出力画像(右)

5.4.3 処理時間の比較

従来手法と提案手法の処理時間の平均を表 5.1 に示す. 処理時間のサンプル数は各 200 個とした.

表 5.1 より目標物体のみを配置した場合, 複数の物体を配置した場合共に提案手法の方がわずかに早いことがわかる. フィルタを作成する際に従来手法では RGB カメラから得た画像をグレースケールに変換する処理をしたが, 提案手法では近赤外線距離画像センサから得た距離画像をそのままフィルタ作成に用いたことによって処理時間の差が出たと思われる.

以上より, 提案手法の方が処理が早く, リアルタイム処理に適していると言える.

表 5.1 従来手法と提案手法の平均処理時間

	目標物体のみ配置	複数物体配置
従来手法	16.02[ms]	15.45[ms]
提案手法	14.80[ms]	15.01[ms]

第 6 章　おわりに

本論文では 3D センサが搭載されている Kinect による距離画像を用いたフィルタを作成し，RGB 画像とあわせた画像分割システムを作成した．目標物体のみが中心に配置された簡単な画像の場合，目標物体を含む複数の物体が配置された複雑な画像の場合共に従来手法より提案手法の方が目標物体の抽出を正確に行え，処理時間も早かった．このため，領域分割システムに提案手法は有効であるといえる．

しかし，従来手法の出力結果は非常に安定していたのに対し，提案手法の出力結果は目標物体全体が切り出せても出力画像が比較的不安定であった．これは，近赤外線距離画像センサのノイズの影響によるものと思われる．

今後の課題として，距離情報と RGB 情報で領域分割された画像を用いて，中心物体の形状，サイズ，テクスチャなど特徴を用いた物体認識システムの特徴抽出部分の研究を進める．また，今回 Kinect の近赤外線距離画像センサから取得した距離画像は，ノイズが大きく安定した領域分割が行えなかったため，ノイズ除去の必要がある．さらに，本システムは中心物体の抽出距離が固定されているため，物体がその範囲外にある場合に対応が出来ない．そのため，中心物体までの距離に応じて切り出すといった開発を進める．

謝辞

本論文作成に当たって日々ご指導頂いた三好力教授には深く感謝いたします. また, 様々なアドバイスや相談を下さった同三好研究室の皆様や, 友人の皆様に深く感謝いたします.

参考文献

[1] “画像内の領域分割と統合”

http://cryst.tv/experiment/ImageProcessing/DomainDivisionAndIntegration/

[2] “凝集法とk-means 法”

http://www.is.doshisha.ac.jp/report/2008/7/5/20081014002/index.html

[3] 岡田和典, “ミーンシフトの原理と応用”, サンフランシスコ州立大学(2009.9)

[4] “エッジ強調”

http://www29.atwiki.jp/suffix/

[5] 佐藤元昭, “一般物体認識における画像の学習効率および認識率向上に関する検討”, 早稲田大学(2010.2)

[6] “Kinect - Xbox.com”

http://www.xbox.com/ja-JP/kinect

[7] 根津禎, “Kinectに見るジェスチャー入力の可能性”, 日経エレクトロニクス（2010.12.27）

[8] “OpenCV - 逆引きリファレンス”

http://opencv.jp/cookbook/index.html

[9] “OpenNIとMS SDK の優劣”

http://itpro.nikkeibp.co.jp/article/COLUMN/20111114/374448/

[10] 中村薫, “Kinect センサープログラミング”, 秀和システム（2011.06.01）

[11] 広瀬大樹, 三好力, “複数センサーを用いた画像領域分割”, 情報処理学会第 75 回全国大会（2013 発表予定）

付録

ソースコード

```
#include <opencv2/opencv.hpp>
#ifdef _DEBUG
//Debugモードの場合
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_core220d.lib")
// opencv_core
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_imgproc220d.lib")
// opencv_imgproc
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_highgui220d.lib")
// opencv_highgui
#else
    //Releaseモードの場合
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_core220.lib")
// opencv_core
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_imgproc220.lib")
// opencv_imgproc
    #pragma
comment(lib,"C:\\OpenCV2.2\\lib\\opencv_highgui220.lib")
// opencv_highgui
#endif
#include <XnCppWrapper.h>
#include <time.h>
#pragma comment(lib,"C:/Program
files/OpenNI/Lib/openNI.lib")

#define SAMPLE_XML_PATH "C:/Program
Files/OpenNI/Data/SamplesConfig.xml"
using namespace cv;
using namespace xn;

int main()
{
    int i,j,k;
    int f_counter = 0;
    clock_t oldTime, newTime;

    int bright_min,bright_max;
    int bright_avg;

    //FILE *outputfile;         // 出力ストリーム
    //outputfile = fopen("data200画素切り出し(中心青イス、後藍
ベッド).txt", "a");  // ファイルを書き込み用にオープン(開く)


    //const int cluster_count = 4;

    //OpenNI
    DepthGenerator depthGenerator;
    ImageGenerator imageGenerator;
    //UserGenerator userGenerator;
    DepthMetaData depthMD;
    ImageMetaData imageMD;
    SceneMetaData sceneMD;
    Context context;

    //OpenCV
    Mat image(480,640,CV_8UC3);             //RGB画像
    Mat depth(480,640,CV_16UC1);        //深度画像

    Mat mask(480,640,CV_16UC1);

    Mat mask1(480,640,CV_8UC1);                    //距離取得失
敗画像...①
    Mat mask2(480,640,CV_8UC1);                    //指定距離取
得成功画像...②
    Mat mask3(480,640,CV_8UC1);                    //指定外距離
取得成功画像...③

    Mat initmask(480,640,CV_8UC3);                 //マスク画像
初期化用

    Mat bgmask(480,640,CV_8UC3);                   //①のカラー
画像
    Mat midmask(480,640,CV_8UC3);                  //②のカラー
画像
    //Mat nearmask(480,640,CV_8UC3);    //③のカラー画像

    Mat bgMask(480,640,CV_16UC1);                  //①の二値化
画像
    Mat midMask(480,640,CV_16UC1);                 //②の二値化
画像
    //Mat nearMask(480,640,CV_16UC1);   //③の二値化画像

    //Mat player(480,640,CV_8UC3);         //人間画像
    //Mat playerDepth(480,640,CV_16UC1);    //人間深度画像

    //Mat background(480,640,CV_8UC3);     //背景画像
    //Mat bgdepth(480,640,CV_16UC1);              //深度背景画
像

    //Mat useMask(480,640,CV_16UC1);         //ユーザマスク


    Mat tmp_img1(480,640,CV_8UC3);
    Mat tmp_img2(480,640,CV_8UC3);

    Mat dst_img1(480,640,CV_8UC3);
    Mat dst_img2(480,640,CV_8UC3);
    //Mat canny_img(480,640,CV_8UC1);

    Mat tmp_dst1(480,640,CV_8UC1);
    Mat imgtmp1(480,640,CV_16UC1);

    Mat tmp_dst2(480,640,CV_16UC1);
    Mat imgtmp2(480,640,CV_16UC1);

    Mat canny_img;

    Mat bin_img;    //RGBイメージの2値化画像用
    vector< vector<cv::Point> > contours;
    Mat contourImage(480,640, CV_8U, Scalar(255));
    Mat init_contourImage(480,640, CV_8U, Scalar(255));
    const int drawAllContours = -1;

    Mat bright_mask(480,640,CV_8UC1);
    Mat bright_Mask(480,640,CV_8UC3);
    Mat rgbmask(480,640,CV_8UC3);

    Mat depth_bg(480,640,CV_16UC1);         //深度背景画像

    int maskSize = mask.step * mask.rows;    //マスク画像の配
列数
    int maskSize2 = mask1.step * mask1.rows;
    int maskSize3 = bgmask.step * bgmask.rows;


    //OpenNIの初期化
    context.InitFromXmlFile(SAMPLE_XML_PATH);
    context.FindExistingNode(XN_NODE_TYPE_DEPTH,
depthGenerator);
    context.FindExistingNode(XN_NODE_TYPE_IMAGE,
imageGenerator);
    //context.FindExistingNode(XN_NODE_TYPE_USER,
userGenerator);

    //RGB画像と振動画像のズレを補正

depthGenerator.GetAlternativeViewPointCap().SetViewPoint(ima
geGenerator);

    //ウィンドウの準備
    cvNamedWindow("image");
    //cvNamedWindow("depth");

    //cvNamedWindow("player");
    //cvNamedWindow("playerDepth");

    //cvNamedWindow("background");
    //cvNamedWindow("bgdepth");


    //cvNamedWindow("mask1");
    cvNamedWindow("mask2");
    cvNamedWindow("midmask");

    cvNamedWindow("laplace1");
    cvNamedWindow("bin");

    cvNamedWindow("canny");
    cvNamedWindow("Contour Image");
    /*cvNamedWindow("k-means");*/

    unsigned short* dp = (unsigned short*)depth.data;
    unsigned short* dp_bg = (unsigned short*)depth_bg.data;
    short tmp_dp;

    int key = 0;
    while (key!='q'){
        oldTime = clock();

        context.WaitAndUpdateAll();

        imageGenerator.GetMetaData(imageMD);
        //depthGenerator.GetMetaData(depthMD);    //距離取得
```

```
        //userGenerator.GetUserPixels(0,sceneMD);    //ユーザピクセル取得

        memcpy(image.data,imageMD.Data(),image.step * image.rows);    //イメージデータを格納
        //memcpy(depth.data,depthMD.Data(),depth.step * depth.rows);    //深度データを格納

        //if(f_counter == 0){
        //
    memcpy(depth_bg.data,depthMD.Data(),depth_bg.step * depth_bg.rows);
        //}

        //for(k=0,i=0;i<mask1.rows;i++){
        //  for(j=0;j<mask1.cols;j++,k++){
        //          tmp_dp = dp_bg[k] - dp[k];
        //          if(tmp_dp > 50){
        //                  if(dp[k] < 1800 && dp[k] > 1000 ){
        //
    mask2.data[i*mask2.cols + j] = 255;
        //                              //mask3.data[i*mask3.cols + j] = 0;
        //          }
        //          else{
        //                  mask2.data[i*mask2.cols + j] = 0;
        //                              //mask3.data[i*mask3.cols + j] = 0;
        //                  }
        //          }
        //          else{
        //
    mask2.data[i*mask2.cols + j] = 0;
        //                              //mask3.data[i*mask3.cols + j] = 0;
        //          }
        //  }
        //}

        //BGR を RGB へ
        cvtColor(image,image,CV_RGB2BGR);

        //player = 0;                                   //初期化
        //playerDepth = 0;                            //初期化
        //memcpy(mask.data,sceneMD.Data(),maskSize); //マスクデータをコピー
        //mask.convertTo(useMask,CV_8UC1);        //マスクの変換
        //image.copyTo(player,useMask);             //マスクを利用した人物抽出
        //depth.copyTo(playerDepth,useMask);        //マスクを利用した人物奥行き抽出
        //background = image - player;            //背景のみ取得
        //bgdepth = depth - playerDepth;
    //深度画像から背景のみ取得

        //距離データの取得できなかった座標のマスク処理
        //mask1.convertTo(bgMask,CV_8UC1);
        //image.copyTo(bgmask,bgMask);

        ////距離データの取得できた座標のマスク処理
        //mask2.convertTo(midMask,CV_8UC1);
        //image.copyTo(midmask,midMask);

        //mask3.convertTo(nearMask,CV_8UC1);
        //image.copyTo(nearmask,nearMask);

        //RGB→グレースケール
        image.convertTo(imgtmp1,CV_8UC3);
        cvtColor(imgtmp1, tmp_dst1, CV_BGR2GRAY);


        bright_avg = tmp_dst1.data[tmp_dst1.cols * tmp_dst1.rows /2 + tmp_dst1.cols/2];
        bright_min = bright_avg - 50;
        bright_max = bright_avg + 50;
        printf("%d\t%d\n",tmp_dst1.data[tmp_dst1.cols * tmp_dst1.rows / 2],tmp_dst1.data[maskSize2/2]);
        //printf("%d\n",tmp_dst1.data[0]);
        //printf("%d,%d\n",bright_min,bright_max);
        for(k=0,i=0;i<mask2.rows;i++){
            for(j=0;j<mask2.cols;j++,k++){
                    if(tmp_dst1.data[k] >= bright_min && tmp_dst1.data[k] <= bright_max ){

    bright_mask.data[i*bright_mask.cols + j] = 255;
                    }
                    else{

    bright_mask.data[i*bright_mask.cols + j] = 0;
                    }
            }
        }

        bright_mask.convertTo(bright_Mask,CV_8UC1);
        image.copyTo(rgbmask,bright_Mask);

        //微分処理(sobel,laplacian,canny)
        //Sobel(tmp_dst, tmp_img, CV_16U, 1, 1);
        //convertScaleAbs(tmp_img, dst_img1, 1, 0);

        //Laplacian(tmp_dst1, tmp_img1, CV_16U, 1);
        //convertScaleAbs(tmp_img1, dst_img1, 1, 0);

        //Canny(tmp_dst1, canny_img, 50, 200);

        //threshold(tmp_dst1, bin_img, 0, 255, THRESH_BINARY|THRESH_OTSU);

        //findContours(bin_img, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_NONE);

        //drawContours(contourImage, contours, drawAllContours, Scalar(0),CV_FILLED);

        //Laplacian(tmp_dst2, tmp_img2, CV_8U, 3);
        //convertScaleAbs(tmp_img2, dst_img2, 1, 0);

        //Canny(tmp_dst, canny_img, 50, 200);


        //画面に表示
        imshow("image",image);
        //imshow("depth",depth);
        //imshow("player",player);
        //imshow("playerDepth",playerDepth);
        //imshow("background",background);
        //imshow("bgdepth",bgdepth);

        //imshow("mask2",midmask);
        //imshow("midmask",midMask);

        //imshow("laplace1",dst_img1);

        imshow("bin",bright_mask);
        imshow("canny",rgbmask);

        //imshow("grayscale", tmp_dst1);

        memcpy(bgmask.data,initmask.data,maskSize3);
        //memcpy(nearmask.data,initmask.data,maskSize3);
        memcpy(midmask.data,initmask.data,maskSize3);
        memcpy(rgbmask.data,initmask.data,maskSize3);


memcpy(contourImage.data,init_contourImage.data,maskSize2);

        newTime = clock();

        //printf("clock [sec]      = %.3f \n", ((float)newTime - (float)oldTime) / (float)CLOCKS_PER_SEC);
        //if(f_counter > 10 && f_counter < 210){
        //  fprintf(outputfile, "clock [sec]      = %.3f \n",((float)newTime - (float)oldTime) / (float)CLOCKS_PER_SEC); // ファイルに書く
        //}
        //else if(f_counter == 210){
        //  fclose(outputfile);
        //  printf("end\n");
        //}
        f_counter++;
        key = waitKey(33);
    }
    context.Shutdown();
    return 0;
}
```