Vol.2013-OS-126 No.2
2013/7/31

# GPGPU を用いた 3D スキャナ向けメモリ管理手法

松本達弥[†1]　藤田悟[†2]

KinectFusion の手法に代表される GPGPU を利用したリアルタイムな 3D スキャナのための効率的なメモリ管理手法について述べる．この手法を利用したスキャナは，取得可能な空間の広さ・細密度が GPU 用のメモリサイズに制限される．この課題を解決するために, GPGPU 向け Octree 構造を利用した効率的なデータ格納・処理手法が既に提案されている．本研究では，ある時点で不必要なデータを，GPU 用のビデオメモリからメインメモリへ退避させ，再度，必要となった際に復帰させる手法，及び，処理効率を向上させるためのデータの並び替えなどを上記の手法に追加し，効率と性能のさらなる向上を試み，メモリ転送量・処理負荷などの視点から，実験と評価を行った．

# Memory Management Method for 3D Scanner Using GPGPU

TATSUYA MATSUMOTO[†1]　SATORU FUJITA[†2]

This paper proposes an efficient memory management system for real time 3D scanner using GPGPU and Kinect. 3D scanners proposed in KinectFusion has limitations on density and range from video memory size for GPU. In order to loosen the limitation, octree is said to work well in terms of data storing and processing. In addition to the original octree approach, we propose a new method which improves efficiency using main memory and sorting octree data and evaluate it from viewpoints of memory transfer size and processing load.

## 1. はじめに

専用の機器を使わない 3D スキャナを実現する手法は，ステレオカメラを利用した方式，複数の視点からの画像を合成する方式などがあり，古くから研究されてきた．しかし，リアルタイムかつ，安価な機器により実現することが難しいという課題があった．この課題を解決することを目的とした比較的安価で精度が高く，かつリアルタイムに動作する 3D スキャナを実現する手法として，Newcombe らによる KinectFusion[1][2]が提案されている．一般家庭での普及が進んでいる家庭用ゲーム機向けモーションセンサであり，Structured Light 方式により深度が測定出来る Kinect と，比較的安価に強力な並列演算性能を得る事ができる GPGPU 技術を組み合わせる事により, 安価でリアルタイムに動作する 3D スキャナを実現する手法である．フリーハンドで Kinect を動かすことを許容しているので，リアルタイムにスキャン状況を確認しながら，特殊スキルや経験なども必要なく，専門的な知識のない一般ユーザでも容易にスキャンを行うことが出来る．

KinectFusion は，Kinect により様々な位置・角度から撮影されたノイズやデータの欠落を含んだ深度マップを, 処理・蓄積し，高精度な物体の表面の推定を行う．色情報や加速度センサなどを利用せず，深度情報だけを用いて処理を行うため，太陽光に含まれる赤外線の影響を受けない室内ならば，周辺の光源など外部の状況に依存せずにスキャンを行うことが出来る．

しかし, 処理対象のボクセルデータを，すべて GPU 用のビデオメモリに保持することが前提の手法であり, かつ，密度を均一に，深さ情報などを保存しているため，何も物体が存在しない空間，物体の内部など，物体表面の形状を取得するためには，不必要な無駄なデータも保持してしまうという問題がある．この問題を解決するために，GPGPU 向けに最適化されたアルゴリズムによる Octree（八分木）構造を利用した高効率な空間データの格納法[3][4], メインメモリを併用する手法[5][6], メインメモリにボクセルではなく一部をメッシュで保持する手法 [6]が既に提案されている．

本論では, 特に Zeng らの Octree の手法に改良を加える．ビデオメモリ消費量を減らし，より広範囲に物体の形状を取得できるようにするため，新たに，ある時点で不必要なデータをメインメモリに退避させ，再度，必要となった時にビデオメモリへ復帰させる．また，データの追加・削除を繰り返すと，メモリ上のデータの位置と３次元空間上の位置に関連性がなくなってしまうことから，データを座標情報によりソーティングすることにより，メモリアクセスを最適化し，高速化を図る．以上の機能拡張について，いくつかの条件が異なるスキャン対象・条件を設定し，実験・評価を行った．

## 2. 関連研究

### 2.1 KinectFusion

KinectFusion[1]は，Kinect で取得した深度マップを GPGPU 技術によって処理し，物体の 3D 形状を取得できる

†1 法政大学 大学院 情報科学研究科
Graduate School of Computer and Information Sciences, Hosei University
†2 法政大学 情報科学部
Faculty of Computer and Information Sciences, Hosei University

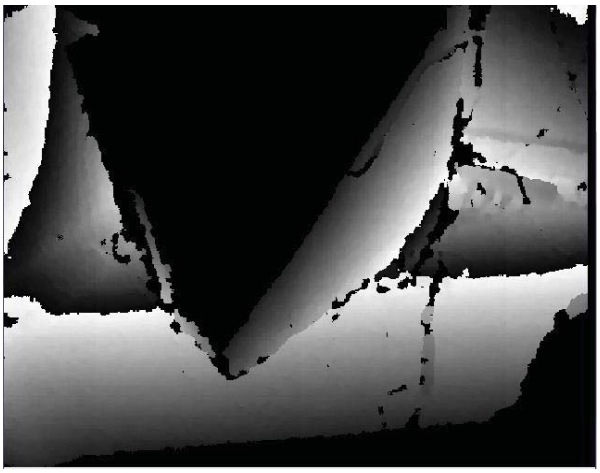

図 1 Kinect で撮影した深度マップ

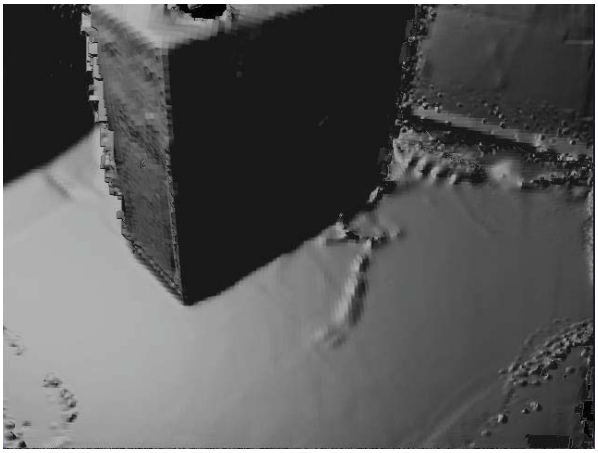

図 2 KinectFusion 方式でスキャンした形状

安価な 3D スキャナを実現する手法である（図 1, 2 参照）．処理は，深度の測定・Kinect の位置・姿勢推定・ボクセルデータの更新・物体の表面推定の主に 4 つのステップに分けられる．

深度の測定では，ノイズや測定距離範囲外の領域に，データの欠落による穴が含まれる Kinect の深度マップ（図 1 参照）をバイラテラルフィルタによって処理する．バイラテラルフィルタは，入力された画像のエッジ情報を保ちながら，ノイズを除去するフィルタである．これにより，姿勢・位置推定の精度を向上させることが出来る．また，フィルタリングされた深度マップから，法線マップを生成する．

Kinect の 6 自由度の位置及び向きの推定には，ICP（Iterative Closest Point）アルゴリズムを利用する．ICP は過去のフレームと現在のフレーム中の点群をマッチングさせ，反復的に比較していくことにより，現在の位置・姿勢を推定する手法である．KinectFusion における ICP の利用法の特徴は，1 フレーム前に撮影した深度マップと比較するのではなく，ボクセルに蓄積した深度情報からフレームを生成し，現在撮影されたフレームと比較する点である．

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | **2** | **1** | **1** | **1** | **2** | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **2** | **1** | **0** | **-1** | **0** | **1** | **2** | 0 | 0 | 0 |
| 0 | 0 | 0 | **2** | **1** | **0** | **-1** | **-2** | **-1** | **0** | **1** | 2 | 0 | 0 |
| 0 | 0 | 0 | **2** | **1** | **-1** | **-2** | 0 | **-2** | **-1** | **1** | 2 | 0 | 0 |
| 0 | 0 | 0 | **2** | **1** | **-1** | **-2** | 0 | **-2** | **-1** | **1** | 2 | 0 | 0 |
| 0 | 0 | 0 | **2** | **1** | **0** | **-1** | **-2** | **-1** | **0** | **1** | 2 | 0 | 0 |
| 0 | 0 | 0 | 0 | **2** | **1** | **0** | **-1** | **0** | **1** | **2** | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | **2** | **1** | **1** | **1** | **2** | 0 | 0 | 0 | 0 |

図 3 Truncated Signed Distance Field

ICP アルゴリズムでは，処理負荷を下げるために，ランダム，もしくは任意にサンプリングした点群同士での比較を行う方法が一般的である．しかし，KinectFusion では，GPU の性能を利用し，取得できたすべての点に対して，マッチングと比較操作を行い，高精度な位置・姿勢推定を実現している．ICP 処理の結果として，カメラ座標系から，ボクセルが格納されている座標系への変換行列を得る．

ボクセルデータの更新では，推定された Kinect の姿勢・位置と深度マップ・法線マップの情報から，ボクセルボリュームを更新する．深度マップはディスプレイ座標系であるため，これをカメラ座標系に変換するため，Kinect 深度カメラ固有のキャリブレーション行列の逆行列を掛ける．ボクセルデータは TSDF（Truncated Signed Distance Field）という値として格納される．実際には更新時の重みのために Weight という値とペアで格納される．TSDF 値は物体表面からの符号付き距離表現である Signed Distance Field をある一定の範囲に限定したものである（図 3 参照）．すなわち，物体表面付近が 0 となり，表面から一定距離内の外側・内側が正，または負の距離となる．また，何も存在しない空間については，0 のまま変化しない．物体の表面推定では，ray-marching 法をボクセルボリュームに対して行い，TSDF の符号が逆転する点を探索する．ray-marching 法は，ray を始点から少しずつ進め，オブジェクトと交差する点を求める手法である．この時，一度にボクセルボリューム内を進む際の大きさを， 何もない空間では，進む量を増やし，TSDF の範囲内にはいったことを検知した後は，減らすことで，高速化を行なっている． 最後に，表面推定により生成された深度マップをシェーディング処理し，スキャン中の形状をユーザに提示する（図 2 参照）．

### 2.2 Octree KinectFusion

Zeng らによって提案された KinectFusion のボクセルデータを Octree 構造で保存する手法である．Octree は子を 8 個もつ 8 分木の木構造であり，X・Y・Z 軸のそれぞれの方向に 2 分割されている．そして，何もない空間については浅

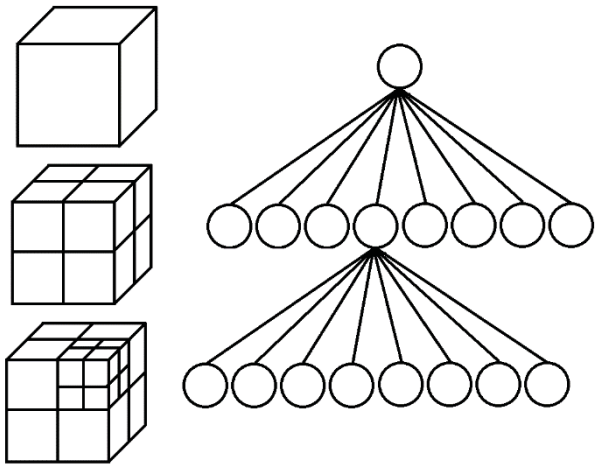

図 4 Octree

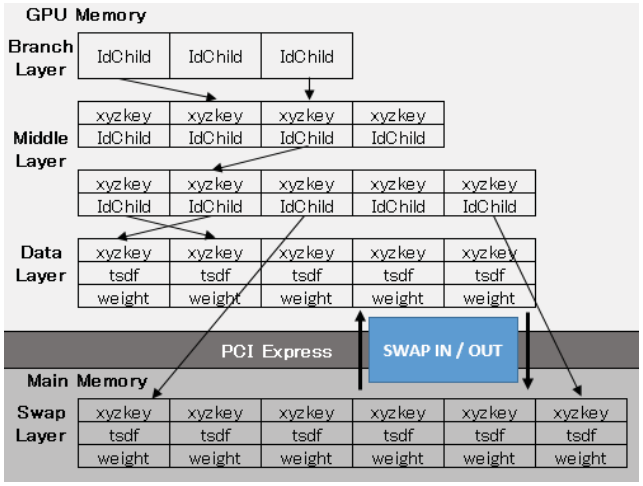


図 5 Octree KinectFusion と SwapLayer のデータ構造

い木の高さ，物体の表面など情報量が多い空間については，深い木の高さにする（図 4 参照）．このことにより，均一なボクセルボリュームとして保存するよりも，効率的に 3 次元物体表面の TSDF を格納することが出来る．また，提案手法では，子を増減させる操作について，GPGPU での実行に特化した並列性の高い手法を利用しており，高速かつボトルネックの少ない操作を実現している．ただし，均一なボクセルボリュームと比較すると，raycast 時，直接，TSDF などの値を読むことが出来ず，子を探索するためのテーブルを木の深さ分，複数回参照する必要がある．その結果として，メモリの参照回数が増加し，処理速度の低下が発生する．このため，木の深さが浅い，大きく何もない空間については，ray-marching 操作の進める長さを，通常よりも大きくする bravely-raymarching の採用が提案されている．結果，オリジナルの KinectFusion と比較し，10%以下のメモリ使用量を実現し，さらに，物体表面推定と TSDF 更新操作において，2 倍以上の高速化を実現している．

## 3. 提案手法

### 3.1 手法の概要

Zeng らの Octree を使用した KinectFusion の手法（以降，Octree KinectFusion と呼ぶ）に対して，一般的な OS のメモリ管理機能を類似したメインメモリを併用する機能（以降，スワップ機能と呼ぶ）とノードのソーティング機能を追加することを提案する．

### 3.2 Octree KinectFusion

#### 3.2.1 データ構造

Octree KinectFusion では，4 種類のデータ階層を持っている．一番上から，Top Layer，Branch Layer，Middle Layer，Data Layer である．Top Layer は raycast 操作を高速化するための層であるが，今回は利用しない．よって，最上位層は Branch Layer である（図 5 参照）．この層は子の位置番号（IdChild）を所持し，すべてのノードが確保されている．なお，子を持たない場合には，IdChild には-1 を格納する．また，インデックス番号が xyzkey というグローバル座標系での位置をエンコードした値となっている．D をノードの深さとした時，xyzkey は式（1）で算出する．

$$\text{xyzkey} = x_1 y_1 z_1 x_2 y_2 z_2 \ldots x_D y_D z_D \qquad (1)$$

次の層である Middle Layer では，すべてのノードが確保されない．この層は，先ほどの IdChild と xyzkey を変数として所持している．また，この層は唯一，任意の数の層を設定することが出来る．

最も深く，木構造の葉部分に相当する Data Layer は 3 種類の変数，xyzkey，tsdf，weight を所持している．

#### 3.2.2 スプリット操作

Octree KinectFusion で子を増やす操作をスプリットと呼ぶ．ノードの中心点と深度マップ上のピクセルを比較し，スプリットを発生させると判断した場合，スプリット判定用フラグを立てる（図 6 6〜11 行目）．その後，Scan 操作[8]を行い，追加するインデックスを生成し（図 6 19 行目），実際に追加処理を行う（図 6 22〜43 行目）

#### 3.2.3 子の削除操作

Octree KinectFusion でノードを消す操作をリムーブと呼ぶ．リムーブを行う条件は，ノードの子が Data Layer の場合は，TSDF がすべての子で有用なデータでないことを表す 0 であること，Middle Layer の場合は，すべての子の IdChild が子を持たないことを表す-1 であることが条件である．

リムーブを発生させると判断した場合，どのノードを消去するかを設定するマージ判定用フラグを立てる（図 7 6〜11 行目）．その後，Scan 操作により，マージ後のインデックス番号を修正するためのデータを生成し（図 7 14 行

```
01: count = number of nodes at L
02: splitFlag[1..count] = 0
03: swapFlag[1..count] = 0
04:
05: // Split・SwapIn する必要があるノードを判定
06: for all node O(i) at Level L in parallel do
07:   if O(i) is in the view frustum then
08:     sdf ← calculate sdf from O(i) and current depth image plane
09:     if (O(i) has no child or has swap) and NeedSplit(O(i), sdf) then
10:       splitFlag[i] = 1
11:       if O(i) has swap then
12:         swapFlag[i] = 1
13:       endif
14:     end if
15:   end if
16: end for
17:
18: // スキャン操作を行う
19: (shift, count) ← Scan(splitFlag, +)
20:
21: // 実際に Split する操作，もしくは，SwapIn を行う
22: for all node O(i) at level L in parallel do
23:   if splitFlag[i] == 1 then
24:     swap_id ← -(O(i).idChild + SwapLayerIdBias)
25:     O(i).idChild ← count + (shift[i] * 8)
26:     key ← O(i).xyzkey * 8
27:     for k from 0 to 7 do
28:       addr ← O(i).idChild + k
29:       if HasSwap(O(i)) then
30:         oct[L + 1][addr] = SwapLayerO[swap_id + k]
31:       else
32:         oct[L + 1][addr].xyzkey ← key | k
33:       endif
34:     end for
35:   end if
36: end for
37:
38: // SwapLayer の開いた穴を埋める
39: (SwapLayer, swapLayerCount) ← Compact(SwapLayer, swapFlag)
40:
41: // 個数を更新
42: O.count = O.count + count * 8
43: SwapLayer.count = swapLayerCount
```

図 6　Octree の追加操作・スワップインの抽象コード

目)，残ったノードの IdChild を変更した後（図 7 17〜36 行目)，マージにより，開いた隙間の部分を Compact 操作[8]で埋める（図 7 39 行目)．

### 3.3 スワップ操作

#### 3.3.1 概要

Octree KinectFusion に本論のスワップ手法を追加するために，さらに，Swap Layer というデータ領域を追加する．この層は 1 層であり，格納している値は Data Layer と同じである（図 5 参照)．上記の層との大きな違いは，他の層はビデオメモリにデータ領域を確保するのに対して，メインメモリに対して確保する点である．実際には，メインメモリに確保した上で，メインメモリ領域の一部を，ビデオメモリとしてマッピングする仕組み[7]を利用し, GPU 上で動作するコードから，ビデオメモリと同じようにアクセス出来るようにしている. また, 今回の提案手法では, Data Layer 上に存在するノードのみをスワップ対象とする．

また，以降，データのビデオメモリから，メインメモリへの退避をスワップアウト，逆をスワップインと呼ぶ.

#### 3.3.2 スワップイン

スワップイン操作は，スプリット操作に処理を追加することで実現する．具体的には，スワップするかどうかの判定をし（図 6 9〜13 行目)，スプリットと同様にフラグを立てる．この時の条件は，深度カメラがノードを捉えているかである．具体的には，ノードの 3 次元空間上の位置をスクリーン座標系に変換した後，そのノードがスクリーンの範囲内かどうかを判定する．その後，Swap Layer 上のインデックスを IdChild から求めて，Data Layer へコピーする（図 6 24〜35 行目)．IdChild に格納されている Swap Layer への Index は式（2）で変換されている．

$$\mathrm{IdChild} = -(\mathrm{SwapLayerChildId} + 2) \qquad (2)$$

但し，SwapLayerChildIdは Swap Layer 上のインデックス番号である．2 を加算しているのは，-1 の値が子を所持していないことを表す特別な番号のため，それと重複することを防ぐためのバイアス値である．

#### 3.3.3 スワップアウト

スワップアウト操作は，リムーブ操作に処理を追加することで実現する. スワップアウトするかどうかを判定し（図 7 8〜9 行目)，リムーブ操作と同様にフラグを立てる．この時の条件は，スワップイン条件とは逆，すなわち，深度カメラがノードを捉えていないことである．そして，リムーブ操作時の IdChild 更新処理時に，Swap Layer へコピーを行う（図 7 25〜29 行目)．この時，元々データが格納されていた親の IdChild に式 2 に基づき，Swap Layer 用 ID を格納する．

#### 3.3.4 スワップアウトのマージン

スワップアウトする条件にマージンを追加する．通常，スワップアウト条件の判定時には，現在，深度カメラが捉えていないノードを対象とする．しかし，この条件の時，カメラに対して，手振れ等による振動が発生する場合，振動による視点の移動の度にスワップが発生してしまう．これを防ぐために，マージンを設ける．通常は，ノードの 3 次元空間上の位置をスクリーン座標系に変換した後，そのノードがスクリーンの範囲内かどうかで，カメラがノードを捉えているかを判定しているが，その範囲を拡張する．例えば，通常は，横 0〜640，縦 0〜480 の範囲を，マージンが 20 の場合, それぞれ, 横-20〜660, 縦-20〜500 とする．

```
01: count ← O.count
02: swapCount ← SwapLayer.count
03: mergeFlag[1..count] ← 0
04: swapoutFlag[1..count] ← 0
05: // どのノードを削除・スワップアウトするか決定
06: for all nodes O(i) at level L - 1 in parallel do
07:   if O(i).idxChild >= 0 then
08:     mergeFlag[O(i).idxChild / 8]
← NeedRemove(O(i)) or NeedSwapOut(O(i))
09:     swapoutFlag[O(i).idxChild / 8]
  ← NeedSwapOut(O(i)) and Layer below L is Data Layer
10:   end if
11: end for
12:
13: // Scan 操作
14: (shift, count) ← Scan(mergeFlag, +)
15:(swapoutScanOut,swapoutScanOutCount)
← Scan(swapoutFlag, +)
16:
17: // ID を更新し，スワップアウトを実行する
18: for all node O(i) at level l - 1 in parallel do
19:   idc ← O(i).idxChild
20:   if idc >= 0 and mergeFlag[idc/8] == 0 then
21:     O(i).idxChild ← idc - shift[idc/8] * 8
22:   else
23:     if swapoutFlag[idc / 8] == 1 then
24:       // SwapOut 処理
25:       swap_layer_idc
←swapoutScanOut[idc / 8] * 8 + SwapCount
26:       O(i).idxChild
← -(swap_layer_idc + SwapLayerIdBias)
27:       for child of O(i) nodes C(j) do
28:         SwapLayerO[j] = C(j)
29:       endfor
30:     else
31:       if mergeFlag[idc / 8] == 1 then
32:         O(i).idxChild ← -1
33:       else
34:         O(i).idxChild ← idc
35:     endif
36:   end if
37: end for
38:
39: // 削除した隙間を埋める
40: Compact(oct[l], mergeFlag, shift)
41:
42: // 個数を更新
43: O.count ← O.count - count * 8
44: SwapLayerO.count
← swapCount + swapoutScanOutCount * 8
```

図 7 Octree の削除操作・スワップアウトの抽象コード

```
01: // ノード数の 1/8(親の個数)を求める
02: count ← number of nodes at level L / 8
03:
04: // ソートを行うためのテーブルを作製
05: for i from 1 to count in parallel do
06:   idxs[i] ← i
07:   xyzkeys[i] ← O(i * 8).xyzkey
08: end for
09:
10: // xyzkey に基づきソート
11: (sorted_idxs, sorted_xyzkeys) ← Sort(xyzkeys, idxs)
12:
13: // ソート結果に基づき，値を移動
14: for i from 1 to count in parallel do
15:   src_idx ← sorted_idxs[sorted_xyzkeys];
16:   new_O(i) ← O(sorted_idxs)
17: end for
18:
19: // 作業結果とスワップ
20: Swap(new_O, O)
```

図 8 ソーティング操作の抽象コード

### 3.4 ソーティング

前記したスプリット・リムーブ，及び，スワップイン，スワップアウトを繰り返すと，初期状態ですべてのノードが確保されている Branch Layer を除いて，メモリ上の位置関係と実際の 3 次元空間上の位置関係が全く無関係になってしまう．例えば，実際の空間上では隣接しているデータが，Data Layer 上で先頭ノードと末尾ノードにそれぞれなってしまう可能性が起こりうる．これによるデータへのアクセス速度の低下などを防ぐために，定期的にソーティングを行う．

具体的には，xyzkey に基づきソーティングを行う．実際には，まず，xyzkey と現在の配列のインデックスを 8 で除算したリストを作成する．（図 8 5〜8 行目），そして，上記のリストをソートする．最後に，ソートされたリストに基づき，データの並び替えを行う（図 8 14〜17 行目）．これを Data Layer から最も浅い Middle Layer まで繰り返す．Branch Layer については，ノードの追加削除が発生せず，並び順が乱れることがないことから行わない．

表 1 実験環境のハードウェア・ソフトウェア

| | |
|---|---|
| CPU | Intel Core i7-3930K 3.2GHz |
| メインメモリ | DDR3-SDRAM 8GB |
| GPU | NVIDIA GeForce GTX 690（メモリ: 2GB） |
| 深度カメラ | Kinect for Windows |
| ライブラリ | CUDA SDK 5.0 |
| | Kinect SDK 1.7 |
| | cudpp 2.0 |

## 4. 評価実験

### 4.1 実験環境

実験環境のハードウェア及びソフトウェアは表 1 の通りである．

KinectFusion は Kinect SDK 付属のライブラリとして利用できるようになっているが，ソースコードが公開されていないため，筆者らが独自に Newcombe らの論文を元に実装を行ったものに対して，Zeng らの GPGPU 向け Octree 構造の手法と，本論で提案した手法を追加した．ただし，ICP 処理については，表面推定処理によって生成されたメッシュと深度マップのすべての点同士の深度の比較ではなく，毎回，ランダムに選択した 1024 個の深度マップの点に対応するボクセルボリューム上の深度を raycast 操作で求め，比較を行なっている．また，Zeng らの提案している Octree を利用した raycast の高速化手法である bravely-raycast につい

ては未実装である．よって，動作速度と Kinect の位置・姿勢推定の精度が，先行手法に比較すると劣化してしまっている．そのため，特に動作速度については，その問題点を考慮した上で結果の考察を行う．

なお，Point Clouds による kinfu[9]という KinectFusion のオープンソース版クローンが存在し，いくつかの KinectFusion 手法の改良の提案に利用されている[3][4]が，今回は提案手法の適用の実験対象としない．

Scan・Compact・Sort については，Octree KinectFusion と同様に cudpp[10] を利用している．

### 4.2 実験用データ

実験用データとして，椅子を一方向から撮影した Chair，パーティションやデスクがある環境を，カメラを動かしながら撮影した Room，椅子に座っている人物を周囲から撮影した Human の３種類を用意した．いずれも，Kinect を手で持って撮影しているため，若干の手振れによる映像の揺れが存在する．長さは１分前後であり，Kinect SDK 付属の Kinect Studio により，深度データを保存した．なお，常に Kinect の撮影モードは，遠距離を中心に測定する Far モード（0.5m～4.0m）に設定した．

### 4.3 実験条件

Kinect Studio の録画データ再生機能を利用し，予め撮影した３つの実験用データを再生させ，アルゴリズムや設定値など条件を変えた状態で実験を行った．なお，ボクセル空間の大きさは縦･横･深さのいずれも 512 個に設定した．

実験では，各手法・設定を用いた際のメモリ使用量・処理速度の変化などを 1 秒間隔で記録した．また，いずれの実験も，同一条件で 5 回実験を行い，その平均値を採用した．なお，メモリ消費量については，実際に TSDF 値などが格納されている Data Layer のみを測定した．

### 4.4 実験 1 メインメモリを併用したスワップ

メインメモリを併用しない・メインメモリを併用し，現在，カメラが撮影している部分のデータのみをビデオメモリ上に保持する・同じくカメラが撮影している周囲部分 100 ピクセル分を保持する, 同 200 ピクセル分を保持する，の３条件を用いて実験を行った．

### 4.5 実験 2 xyzkey によるソート

スワップを有効かつ周囲 200 ピクセル分を保持するように設定したものと，更にそれに対して，xyzkey によるソートを有効にしたものの比較実験を行った．比較項目は，深度測定時の深度マップ・法線マップ生成処理，ボクセル更新処理・raycast による表面検出の合計処理時間である．

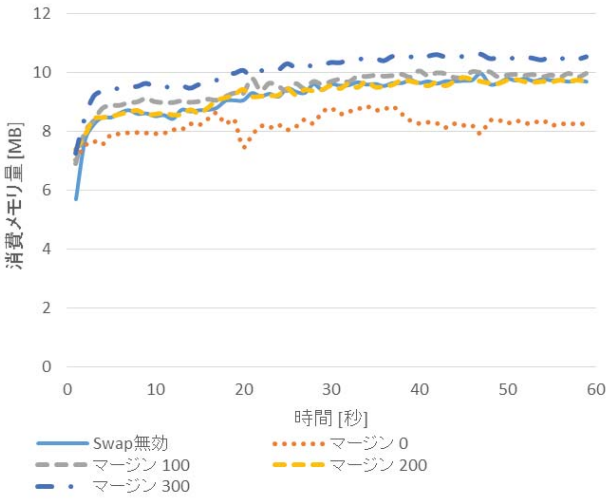


図 8 Chair を対象としたメモリ消費量

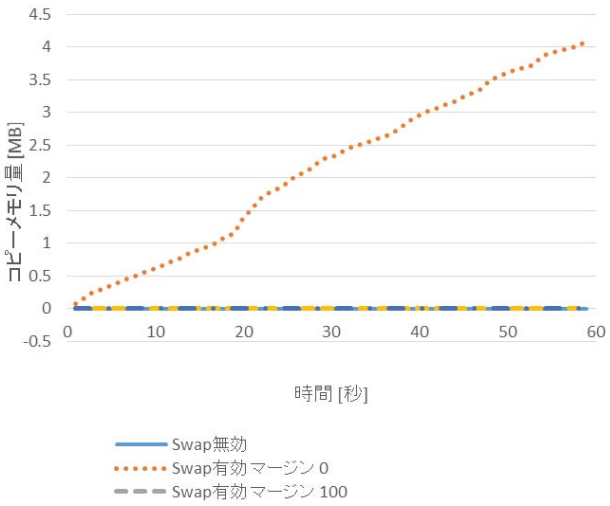


図 9 Chair を対象としたビデオメモリ・メインメモリ間転送量

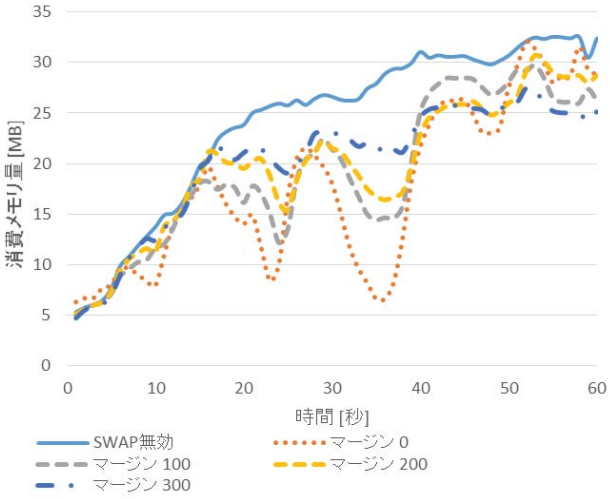


図 10 Room を対象としたメモリ使用量

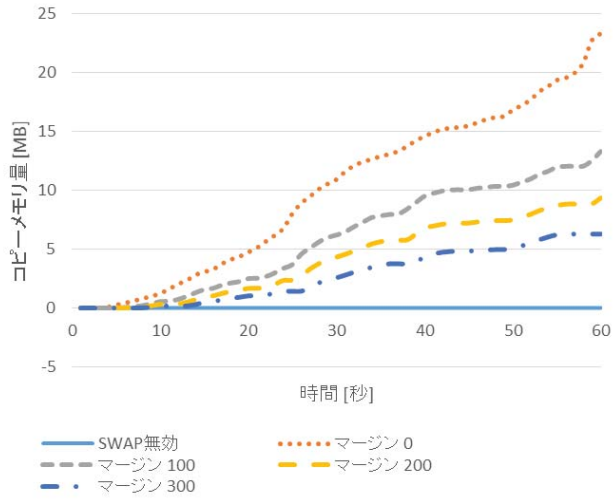


図 11 Room を対象とした対象とした
ビデオメモリ・メインメモリ間転送量

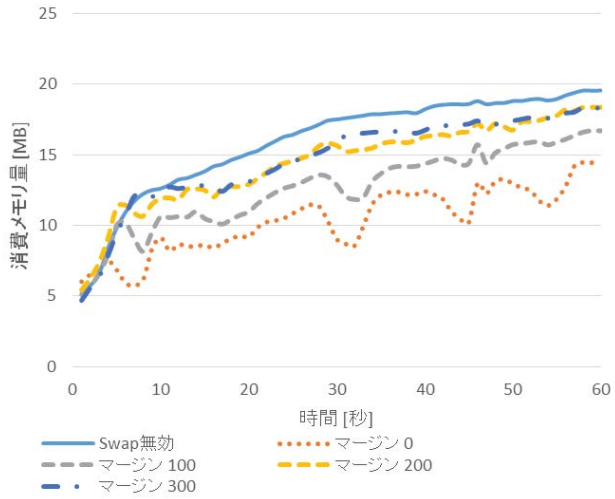


図 12 Human を対象としたメモリ消費量

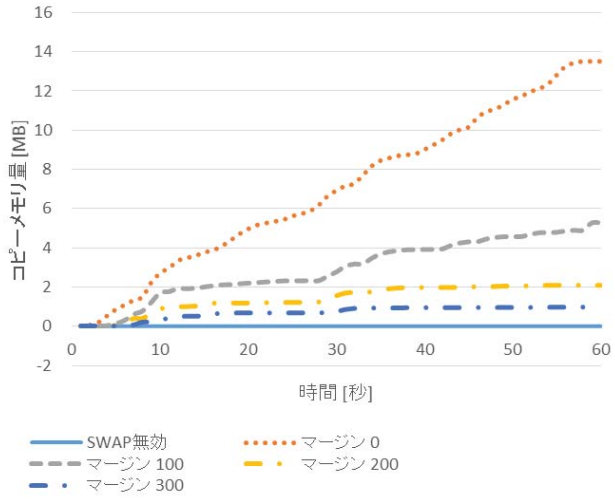


図 13 Human を対象とした対象とした
ビデオメモリ・メインメモリ間転送量

表 2 各データの平均処理時間 [秒]

| | Chair | Room | Human |
|---|---|---|---|
| 無効 | 52.7 | 63.3 | 59.1 |
| マージン 0 | 52.8 | 66.28 | 61.1 |
| マージン 100 | 53 | 64.32 | 60.0 |
| マージン 200 | 52.3 | 66.8 | 60.0 |
| マージン 300 | 53.2 | 65.7 | 60.8 |
| ソート済みマージン 200 | 59.45 | 71.3 | 66.9 |

表 3 処理ステップごとの処理時間の割合

| | 深度測定 | ICP | Update | Raycast | ソート |
|---|---|---|---|---|---|
| ソート有効 | 1% | 67% | 2% | 22% | 0% |
| ソート無効 | 1% | 68% | 2% | 20% | 2% |

## 5. 考察

ビデオメモリ・メインメモリ間転送量については，Chair についてマージン 0 の場合のみに顕著に転送量が経過時間に比例して増加している. Chair はある特定の箇所を常に撮影し続けるデータであるが，手振れなどの影響で若干の視点移動が発生している．マージンを設けないことで，この視点移動によるスワップが発生し，転送量が増加していると考えられる．他のデータに対する実験では，マージンが大きくなるほど，転送量が少なくなっている．ただし，Human など視点移動量が大きくないデータでは，マージン 200 の場合と同 300 の場合で差が他の場合と比較すると少なくなっている．これは，マージンの大きさが 200 の場合で十分であることが原因であると考えられる．

メモリ消費量については，マージンの大きさにある程度比例してメモリ消費量が増加している．ただし，Human などのケースでは，マージンが 200 の場合と 300 の場合でほぼ同じ消費量となっている．これは，マージンがこの場合は 200 で十分であることを表していると考えられる. また，消費量の変動については，マージンの大きさに比例して少なくなっている．マージンが大きければ，手振れや，一定の範囲内の反復移動をした場合に，メインメモリへ退避させる場合などが少ないからであると考えられる．

処理時間については，スワップの有効・無効，マージンの大きさにほぼ関係ない処理時間となっている．これは，実験条件の項で示したとおり，raycast 操作・ICP 処理が最適化出来ておらず，その 2 つがほぼ処理時間の大半を占めており，スワップ処理については，殆ど無視出来る処理時間であることが原因であると考えられる（表 3 参照）.

xyzkey をキーとしたソーティングについては，動作速度

を比較すると，高速化は実現できておらず，むしろ，性能が劣化してしまっている．これは，本来期待した raycast 操作の高速化が予想よりも実現できておらず，かつ，ソーティングの負荷がそれを上回ってしまっていることが原因であると考えられる．

本論の提案手法の一つである KinectFusion とメインメモリの併用は，Heredia らの研究[3]，Whelan らの Kinitinous[4]で既に試みられている．これらの手法と本論の提案手法の違いは，第一に，Kintinous については，メインメモリに退避した TSDF などのボクセルデータを CPU によりメッシュ生成することで，データ量を削減するという先進的な方法を採用しているが，メッシュ化したデータを再度，ビデオメモリ上のボクセルボリュームに再度利用することには，部分的にしか成功していない．第 2 に，どちらの手法も，ビデオメモリ上のデータについては，均一な間隔のボクセル形式を採用しており，本論のように Octree を採用していないことから，ビデオメモリ消費量の削減には成功していない．また，Heredia らの研究については，物体表面などボクセルに有用なデータが含まれている部分のみをメインメモリへコピーしているが，Kintinous については，TSDF 値が 0 の何もない空間についても，メインメモリへのボクセルデータのコピーをしていることから，本論の提案手法などと比較するとビデオメモリ・メインメモリ間の通信帯域を多く消費していると思われる．

ビデオメモリの消費量の削減と利用可能な記憶領域の拡大を効率よく実現出来ている本論の提案手法は，ビデオメモリの消費量の削減，及び，メインメモリへの退避・復帰操作時の帯域の消費量という視点と，それに伴う三次元形状の取得可能な範囲の拡大という点での評価では，先行手法と比較し，より優れていると考える．

## 6. まとめ

本論では，Newcombe らの KinectFusion と Zeng らの KinectFusion 向け Octree 実装を改良し，GPGPU 向けプログラムのパフォーマンスのために重要な並列性を維持しつつ，メインメモリの併用と xyzkey をキーとしたデータのソーティングによる処理時間の高速化を行う手法を提案した．3 種類の異なる環境・条件の元で撮影した深度動画データに対して評価実験を行った結果，メインメモリの併用については，速度を大幅に低下させることなく，ビデオメモリ使用量を削減する効果が確認できた．また，メインメモリへの退避させるデータについては，ディスプレイ座標系において，現在，撮影・表示している周囲 200 ピクセル程度のマージンをとることで，Kinect の手振れなどによる無駄なメインメモリへの退避を防ぐことが出来ることが分かった． xyzkey に基づくデータのソーティングについては有意な処理速度の改善を確認することが出来なかった．

今後の課題として，一つは，ボクセル空間の解像度の一辺の上限が 1024 個に限られてしまっている問題の解決がある．これについては，xyzkey の大きさが 32bit であることが原因である．仮に，一辺当り，2048 個とすると 2 の 11 乗なので，11bit が必要となり，3 辺で合計 33bit 必要となり，32bit を超えてしまうからである．これについては，処理の整合性と性能を保ちながら，xyzkey を 64bit 化することで解決することを現在．筆者らは進めている．

また，SSD や HDD など永続的な記憶媒体への適応も検討課題である．これらは，メインメモリに比べると低速であるが，記憶容量が大きいため，うまく適応出来れば，現状のシステムと比較すると大幅に物体の形状を取得できる範囲を広げることが出来る可能性がある．また，複数台の Kinect を利用した場合のデータのマージ等についても検討中である．さらに，現在，メインメモリへ退避させる領域決定にディスプレイ座標系を用いているが，これをカメラ座標系に置き換え，さらに動きの予測を追加することで，回転などの動きに対してより，スワップの回数，及び，転送量を減らすことができる可能性がある．

### 参考文献

1) R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, A. Fitzgibbo：Kinectfusion: Real-time dense surface mapping and tracking, Procedings of IEEE / ACM International Symposium on Mixed and Augmented Reality, pp. 127–136, 2011.

2) S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. A. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, A. Fitzgibbon：KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera, ACM Symposium on User Interface Software and Technology, 2011.

3) M. Zeng, F. Zhao, J. Zheng, X. Liu：Octree-based Fusion for Realtime 3D Reconstruction, Graphical Models, Volume 75, Issue 3, pp. 126‑136, 2013.

4) M. Zeng, F. Zhao, J. Zheng, X. Liu：A memory-efficient kinect-fusion using octree, Proceedings of Computational Visual Media 2012, pp. 234-241, 2012.

5) F. Heredia, R. Favier：KinectFusion extensions to large scale environments, The Point Cloud Library（オンライン），入手先〈http://www.pointclouds.org/blog/srcs/fheredia/index.php〉（参照 2013-06-24）.

6) T. Whelan, J. McDonald, M. Kaess, M. Fallon, H. Johannsson, J. J. Leonard：Kintinuous: Spatially extended kinectfusion, RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras, 2012

7) CUDA C Programming Guide: NVIDIA Corporation（オンライン），入手先〈http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html〉（参照 2013-06-24）.

8) M. Harris, S. Sengupta, John D. Owens：GPU Gems 3, chapter 39, pp. 851‑876, 2007.

9) The Point Clound Libray: kinfuls, The Point Cloud Library（オンライン），入手先〈http://docs.pointclouds.org/trunk/namespacepcl_1_1device_1_1kinfu_1_s.html〉（参照 2013-06-24）.

10) M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, A. Davidson ：cudpp - CUDA Data Parallel Primitives Library，Google Project Hosting（オンライン），入手先〈https://code.google.com/p/cudpp/〉（参照 2013-06-24）.