GPUを利用した高速・高効率な３Dスキャナ

MATSUMOTO, Tatsuya / 松本, 達弥

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
9
(開始ページ / Start Page)
143
(終了ページ / End Page)
148
(発行年 / Year)
2014-03
(URL)
https://doi.org/10.15002/00010536

GPUを利用した高速・高効率な3Dスキャナ
High-performance 3D Scanner using GPU

松本　達弥
Tatsuya Matsumoto
法政大学大学院情報科学研究科情報科学専攻
E-mail: tatsuya.matsumoto.7n@stu.hosei.ac.jp

Abstract

This paper proposes a high-performance real time 3D scanning method using GPU by extending Octree KinectFusion. In the Octree KinectFusion, all data are stored in video memory and a surface generated by a ray marching method can be displayed in real time. To make the data area larger, this paper proposes a method to use not only video memory but main memory. A way to use both main and video memories is similar to a swap system in operation system. When values on video memory get unnecessary for processing, they will be moved to main memory like swap-out and when values on main memory get necessary for processing, they will be back to video memory like swap-in. It costs so high for a program on GPU to exchange values on main memory that the proposed method inserts margins between camera screen and active video memory areas for avoiding excessive swap-in/out operations. To generate a surface faster, this paper uses a marching cube method. A ray marching method used in the previous study generated a depth map. It ran faster than a marching cube method but needed to run for every frame. On the other hand, a marching cube method generates polygons which neither changes often nor needs to run for every frame. Experimental results showed that the proposed method could capture bigger volumes of scenes and processed faster than the previous study and output high quality 3D model.

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や，複数の視点からの画像を合成する方式などが，研究されてきた[1]．しかし，リアルタイムかつ，安価な機器により，密な形状情報の取得を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手法の一つとして，Newcombe らは KinectFusion [2][3]を提案した．これは，家庭用ゲーム機向けモーションセンサであり，約 3m の区間で深度が測定出来る Kinect と，安価に強力な並列演算性能を発揮する GPGPU 技術を組み合わせ，リアルタイムに動作する安価な 3D スキャナを実現する手法である．Kinect を手で持ちながら，リアルタイムにスキャン状況を確認出来ることから，専門知識のない一般ユーザでも容易に三次元スキャンを行う事ができる．

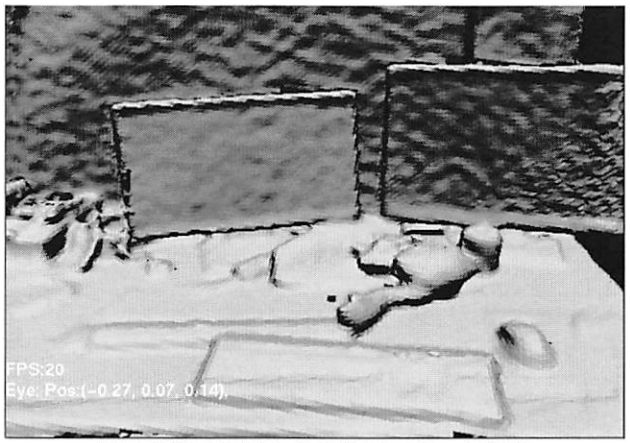

図 1 提案手法による三次元スキャン結果

KinectFusion は，Kinect により様々な位置・角度から撮影された深度マップを，処理・蓄積し，物体表面の推定を行う．この技法は，スキャン対象の空間全体を均一なグリッド状のボクセルボリュームとして，GPU から高速にアクセス出来るビデオメモリ上に保持することが前提の手法である．このため，空間の広さ・保存時の細密度に比例し，ビデオメモリを消費する．また，処理負荷についても，同様に，その大きさに比例して，増大する．よって，広く細密な空間をスキャンするには，大容量ビデオメモリと高速 GPU を搭載した高価なビデオカードが必要となる．

そこで，本論では，安価な機器により，広く細密な空間をスキャンするため，KinectFusion 手法のビデオメモリ消費量の削減と，処理の高速化を実現する手法を提案する．提案手法では Octree（八分木）構造を利用した高効率な KinectFusion 向け深度情報の格納手法[4][5]に改良を加える．すなわち，ある時点で，不要なデータを，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ことで，ビデオメモリの消費量を削減する[8][9]．また，この退避・復帰処理は負荷が大きい処理であるため，過剰に発生しないように管理する．その他，先行研究では，Ray marching 法を毎フレーム実行し，スキャン結果のプレビュー表示とカメラ位置・姿勢推定処理で利用する深度マップの生成を行っていた．これを Marching cubes 法[10]によるポリゴン生成に変更する．深度情報が大きく変化しない限り，ポリゴン情報は再生成しなくても，十分な精度の描画と深度マップ生成が行えるため，システム全体の処理性能向上が期待できる．

Supervisor: Prof. Satoru Fujita

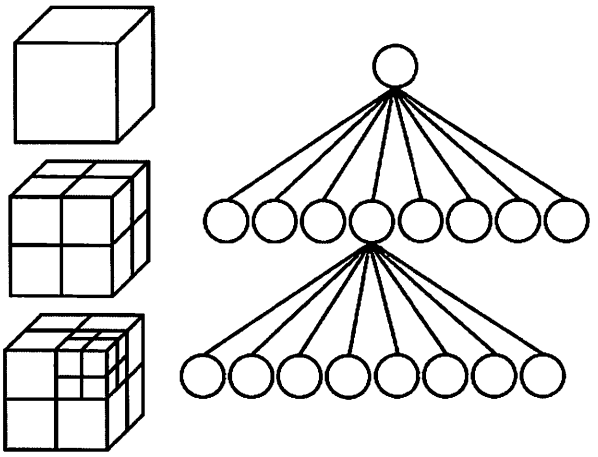

図 2 Octree 構造

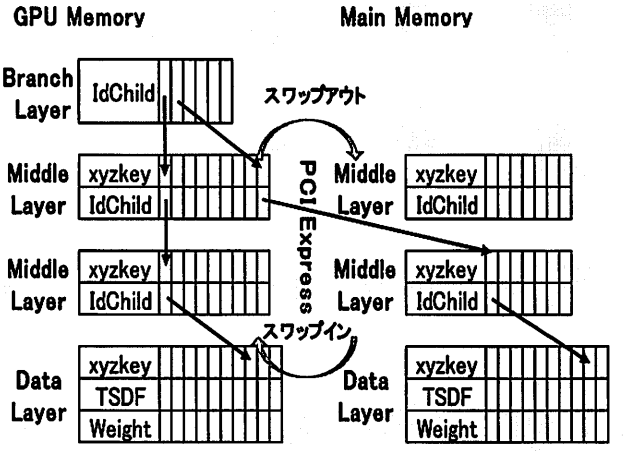

図 3 提案手法のメモリ構造

2. 関連研究

2.1. KinectFusion

KinectFusion 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 形状を取得できる安価な 3D スキャナを実現する手法である．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 の深度マップを，入力された画像のエッジ情報を保ちながら，ノイズを除去するバイラテラルフィルタによって処理する．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[11]を利用する．ICP は過去・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，直接，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，ボクセルボリュームに蓄積した深度情報から相当するフレームを生成し，撮影された現在のフレームと比較する点である．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 の強力な並列演算性能を利用し，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グローバル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り，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 をある一定の範囲で打ち切った値である．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(視線)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．

最後に，表面推定により生成された深度マップ・法線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．

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 2）．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 次元物体表面の深度情報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 での実行に適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，Ray marching 時などに，直接的に，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一番上の層から，木の深さ分，探索処理を行う必要がある．その結果として，メモリの参照回数とプログラム上の分岐回数が増加し，処理速度が低下する．このため，ノードの参照回数を減らし，高速化するため，木の深さが浅い，大きく何もない空間については，Ray marching の進める長さを，大きくする Bravely ray marching 法の採用が提案

```
01: for all node O(i) at Level L in parallel do
02:  if O(i) is in the view frustum then
03:   sdf = calculate sdf from O(i)
            and current depth image plane
04:   splitFlag[i] = (O(i) has no child
                      or has swap)
                      and NeedSplit(O(i), sdf)
05:   swapFlag[i]  = O(i) has swap
06:  end if
07: end for
08:
09: (scanout, count) = Scan(splitFlag, +)
10:
11: for all node O(i) at level L in parallel do
12:  if splitFlag[i] == 1 then
13:   O(i).idChild
       = number of nodes at L + (scanout[i] * 8)
14:   for k from 0 to 7 do
15:    addr = O(i).idChild + k
16:    oct[L + 1][addr].xyzkey
        = (O(i).xyzkey << 3) | k
17:    oct[L + 1][addr]
        = SwapLayer[-(O(i).idChild
          + SwapLayerIdBias) + k]
          if HasSwap(O(i))
18:   end for
19:  end if
20: end for
21:
22: (SwapLayer, swapLayerCount)
     = Compact(SwapLayer, swapFlag)
23:
24: O.count = O.count + count * 8
25: SwapLayer.count
     = SwapLayer.counts - swapLayerCount
```

図4 スプリット・スワップイン擬似コード

されている．結果，オリジナルの KinectFusion と比較し10%以下のメモリ消費量を実現し，さらに，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．

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 Ray marching 法を高速化するための層であるが，本論では利用しない．よって，本論の最上位層は Branch Layer である（図 3 左）．この層は子の位置番号（IdChild）を所持し，位置に対応する全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．

$$xyzkey = x_1 y_1 z_1 x_2 y_2 z_2 \ldots x_D y_D x_D \quad (1)$$

次の層である Middle Layer では，位置に対応する全てのノードが確保する必要がない．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．

2.3. 子の追加(スプリット)

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 4 1～7 行目）．その後，Scan 操作[12]を行い，追加位置インデックスを生成し（図 4 9 行目），実際に対応する場所に追加処理を行う（図 4 11～20 行目）

2.4. 子の削除(リムーブ)

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，どのノードをするかを設定するマージ判定用フラグを立てる（図 5 1～4 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 5 14 行目），残ったノードの IdChild を変更した後（図 5 9～21 行目），マージにより，開いた隙間の部分を Compact 操作[12]で埋める（図 5 22 行目）．

3. 提案手法 1 :スワップシステム

3.1. 概要

Octree KinectFusion に本論のスワップ手法を追加するために，Swap Tree というデータ領域をメインメモリ上に確保する．ルートノードが格納されているため，スワップ出来ない Branch Layer が存在しないことを除いては，ビデオメモリ上と同じメモリ構造が展開されている(図 3 右)．以後，ビデオメモリ上に存在する層に対応する Swap Tree 上の層を Swap Layer と呼ぶ．

また，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機能を利用し，GPU 上で実行されるコードからビデオメモリと同じように，直接アクセスが出来るようにする．

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．

3.2. 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 4 5 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 Tree 上の Layer のインデックスを IdChild から求めて，ビデオメモリ上の対応する Layer へコピーする（図 4 11～20 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．

$$IdChild = -(SwapLayerIdChild + 2) \quad (2)$$

但し，SwapLayerChildId は Swap Layer 上のインデックス

```
01: for all nodes O(i) at level L - 1 in parallel do
02:  mergeFlag[O(i).idChild / 8]
       = O(i).idChild >= 0 and (NeedRemove(O(i))
          or NeedSwapOut(O(i)))
03:  swapoutFlag[O(i).idChild / 8]
        = O(i).idChild >= 0 and NeedSwapOut(O(i))
04: end for
05:
06: (shift, count) = Scan(mergeFlag, +)
07: (swapoutScan, swapoutScansCount)
       = Scan(swapoutFlag, +)
08:
09: for all node O(i) at level l - 1 in parallel do
10:  idc = O(i).idChild
11:  if idc >= 0 and mergeFlag[idc/8] == 0 then
12:   O(i).idChild = idc - shift[idc/8] ? 8
13:  else
14:   if swapoutFlag[idc / 8] == 1 then
15:    swap_layer_idc
          = swapoutScanOut[idc / 8] * 8
             + SwapLayer.count
16:    O(i).idChild = -(swap_layer_idc
                          + SwapLayerIdBias)
17:    Copy nodes from child of O(i) to SwapLayer
18:   else
19:    O(i).idChild
        = mergeFlag[idc / 8] ? -1 : idc
20:  end if
21: end for
22:
23: Compact(oct[l], mergeFlag, shift)
24: O.count = O.count - count * 8
25: SwapLayer.count
     = SwapLayer.count + swapoutScanOutCount * 8
```

図5 リムーブ・スワップアウト擬似コード

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. 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5 3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 5 14～17 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．

3.4. 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，上記の通り，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．

4. 提案手法2:リアルタイムポリゴン生成

4.1. 処理概要

先行手法では，表面推定処理に Ray marching 法，もしくは，Bravely ray marching 法を利用し，深度マップを生成している．これを本論の提案手法では，ある任意の間隔でビデオメモリ上に保存されている深度情報について，Marching cubes 法により，一度，ポリゴンとして出力し，その後，これを次のポリゴン生成までの間，複数フレーム間で使い続け，比較的低負荷な，深度マップを生成する描画処理のみ毎フレーム行い，処理の高速化を行う．以後，この定期的なポリゴン生成処理をフルポリゴン生成と呼ぶ．また，本論での Octree 構造に適した Marching cubes 法の実装と，フルポリゴン生成の間で新規に必要となった部分について，随時．ポリゴン生成を行い，処理のさらなる高速化と深度マップ品質の向上を行う．

4.2. Octree 構造向け Marching cubes 法

Marching cubes 法は深度情報 8 点に基づきポリゴンを生成する．この時，KinectFusion のように均一なグリッド形状でボクセルを格納している場合，深度情報が含まれていない部分も含めて，すべてのボクセルにアクセスし，ポリゴンを生成する必要がある．例えば，ボクセルボリュームのグリッド1辺の大きさが 512 の場合，1億個以上のノードにアクセスする必要があり，リアルタイムな処理は困難である．

本論の提案手法で利用している Octree 構造のように，深度情報が存在する場所だけにノードが存在するデータ構造であれば，アクセスする必要があるノード数を 3%以下まで減少させることができる．また，更に高速化するため，深度情報の取得時に提案手法で採用している GPU 向け Octree 構造の特性を利用し，効率よく実装する．

通常，あるノードの正方向に隣接する 7 ノードを取得する場合，Branch Layer から順番に，木の上部から検索処理を行う．しかし，この処理は 2.2 節で述べた通り，高負荷な処理である．Data Layer のノードは 8 個単位で管理されていることから，1 ノードずつ処理を行うのではなく，8 ノード単位で並列に一度に行うことで処理を高速化できる．8 ノードが処理実行時にアクセスする隣接する別の 8 個ノードのブロックは，常に 8 個である．この性質を用いて，隣接する 8 個のブロックの先頭ノードの位置を求めた後，そこからの相対位置として必要とするノードの位置を算出する実装にすることで，探索処理を 1/8 回に減らすことが出来，高速化できる．

4.3. 差分ポリゴン生成

カメラ視点が移動・回転する場合，フルポリゴン生成時にビデオメモリに存在しなかったノード部分について

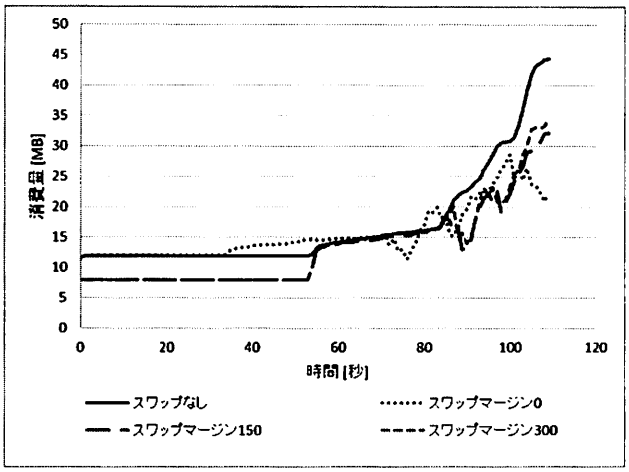

図 7 ビデオメモリ消費量

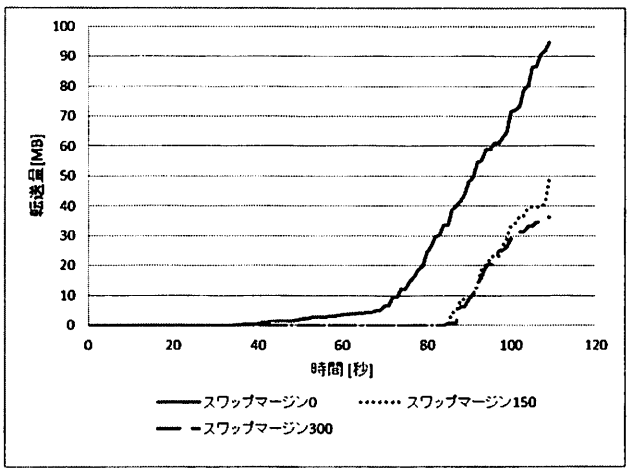

図 8 ビデオ・メインメモリ間転送量

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
深度カメラ SDK	Kinect SDK 1.7
GPGPU SDK	NVIDIA Cuda SDK 5.5
GPGPU ライブラリ	Cudpp 2.0

表 2 1 フレーム当たりの平均処理時間

実装	デプスマップ生成	スワップ	処理時間 [ms]
kinfu	Ray marching		27
本論	Ray marching		66
	Marching cubes		24
	Marching cubes (Optimized)		22
	Marching cubes (Optimized)	○	24

は，ポリゴンが存在しないため表示されない．このため素早くカメラを動かした場合，一時的にポリゴンが表示されない領域が，深度マップの多くを占めることで，ICP 処理が失敗する恐れがある．このため，フルポリゴン生成時から，ビデオメモリへ新規に追加されたノードについては，毎フレーム，別途，ポリゴン生成を行う．

フルポリゴン生成時の Data Layer のノード数を記録しておき，現在のノード数から新規に追加されたノードの範囲を算出しポリゴン生成処理を行う．また，差分ポリゴン生成中に、Data Layer からノードが削除・移動されると矛盾が生じるため，リムーブ処理・スワップアウト処理は，フルポリゴン生成の直前に行う．

5. 実験環境

5.1. 概要

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．

オリジナルの KinectFusion 実装はソースコードが非公開であることから，先行研究の論文を元に独自に実装したものに，本論の提案手法を追加した．ただし，Ray marching については，Bravely ray marching を実装していない．また，比較評価対象として，Point Cloud Library による KinectFusion クローン実装の kinfu[13] を用いる．

5.2. 実験用データ

実験用深度動画データについては，椅子や机などの物体がある環境を，最初はカメラを動かさず，しばらくして，カメラを動かすというように撮影した．なお，カメラは手で持って撮影を行っているため，手振れによる映像の揺らぎが存在する．また，長さは約 2 分である．

5.3. 実験条件

Kinect SDK の録画データ再生機能を利用し，予め撮影した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．

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，同一条件で 5 回実験を行い，その平均値を採用した．

6. 評価実験

6.1. 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，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，スワップマージン 150・300 ピクセルを設定する，の 4 条件を用いて実験を行い，消費ビデオメモリ量とビデオ・メインメモリ間転送量(スワップ量)・処理速度について計測を行った．

6.2. 実験 2: ポリゴン生成による高速化

表面推定処理を Ray casting 法で行ったものと Marching cubes 法で行ったものについて，実験を行い，それぞれの処理速度について計測を行った．

6.3. 実験結果

ビデオメモリ消費量(図 7)については，カメラを動かしている後半では，先行研究である Octree KinectFusion と同等の，スワップしない設定の消費量と比較して，スワ

ップにより，ビデオメモリ消費量が，最大で約 25MB 削減出来ている．また，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．

ビデオ・メインメモリ間転送量(図 8)については，カメラをあまり動かしていない前半部分では，手振れ，もしくは，カメラ姿勢・位置推定の誤差による無駄な転送がマージンによって，最大で約 50MB 程度，防ぐことが出来ている．カメラを動かしている後半については，マージンに関係なく，スワップアウトに伴う転送量がいずれも増加している (表 2).

処理時間については，Ray marching 法を用いたものよりも，Marching cubes 法を用いた提案手法の方が高速に動作している．また，探索処理の高速化を行った Marching cubes 法は未実施のものに比べ，高速に動作している．

7. 考察

本論の提案手法の一つである KinectFusion のメインメモリ併用は， Whelan らの Kintinous[6]，Heredia らの研究[7]で既に試みられている．

Whelan らの Kintinous は，退避する深度情報を，CPU によってメッシュ情報に変換し，メインメモリに保持する．この仕組みにより，ビデオメモリ上に保持するデータ量を最小化しつつ，物体形状を取得する手法を提案している．しかし，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復帰させることには，部分的にしか成功していない．また，有用な深度情報の有無に関わらず，メインメモリへ情報を退避している．よって，本論の提案手法と比較すると，ビデオ・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．

Heredia らの研究は，物体表面など有用なデータが含まれている部分だけ，メインメモリへ退避し，再度，必要となった際に，メインメモリに復帰させる手法を提案している．有用な深度情報が含まれている部分だけ，退避・復帰させるという点では，本論の提案手法と類似している．

しかし，Whelan らの Kintinous，Heredia らの研究，いずれの手法も，ビデオメモリ上のデータ構造として，均一なグリッド状のボクセルボリューム構造を，採用している．よって，本論のように，ビデオメモリ消費量の削減には成功していない．また，ポリゴン生成を利用した深度マップ生成の高速化も，行われていない．

以上，メインメモリを併用する先行手法と比較し，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点，また，表面推定処理の高速化という点による評価では，より優れていると考える．

8. おわりに

本論では，Newcombe らの KinectFusion と Zeng らの Octree KinectFusion を改良し，GPGPU 向けプログラムの性能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 Marching cubes 法によるリアルタイムなポリゴン生成を追加した．評価実験の結果，メインメモリの併用について，処理速度を大幅に低下させず，ビデオメモリ消費量を削減する効果を確認した．また，メインメモリへ退避させるデータについて，判定基準にマージンを設けることで，深度カメラの手振れ等による無駄な退避を防げることを確認した．さらに表面推定については，ポリゴン生成による，高速な処理の実現を確認した．

今後の課題としては，本手法ではほとんど利用していない CPU の活用がある．例えば，データのノイズ除去やスムージング処理を CPU で行うことで，GPU の処理負荷を増やさず，更に高精度な 3 次元モデルの取得が実現出来る可能性がある．

参考文献

1) R. A. Newcombe and A. J. Davison. Live dense reconstruction with a single moving camera. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2010.

2) 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.

3) 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.

4) 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.

5) 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.

6) 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 2014-02-01）.

7) 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.

8) 松本 遠弥, 藤田 悟:GPGPUを用いた3Dスキャナ向けメモリ管理手法, 情報処理学会研究報告, Vol. 2013-OS-126, No. 2, pp.1-8, 2013.

9) 松本 遠弥, 藤田 悟:GPGPUを利用する3Dスキャナ向け高効率メモリ管理方式, 情報処理学会第76回全国大会, 2Q-5, 2014.

10) William E. Lorensen, Harvey E. Cline: Marching Cubes: A high resolution 3D surface construction algorithm, Computer Graphics, Vol. 21, No. 4, 1987.

11) P.J.Besl, N.D.McKay : A Method for registration of 3d shapes, EEE Trans. on Pattern Analysis and Machine Intelligence, Vol14, No2, pp.239-256,1992.

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

13) Point Cloud Library : PCL – Point Cloud Library, The Point Cloud Library（オンライン），入手先〈http://www.pointclouds.org/〉（参照 2014-02-01）.